UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Captura y proceso de datos de aplicaciones, redes y sistemas a alta velocidad

High velocity data capture and processing from applications, networks and systems

Autor: Diego Sáinz de Medrano Otero
Tutor: Sergio Fuentes de Uña
Ponente: Javier Aracil Rico

Julio 2020

EPS - UAM

Captura y proceso de datos de aplicaciones, redes y sistemas a alta velocidad

High velocity data capture and processing from applications, networks and systems

Autor: Diego Sáinz de Medrano Otero
Tutor: Sergio Fuentes de Uña
Ponente: Javier Aracil Rico

ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD AUTÓNOMA DE MADRID

Julio 2020

EPS - UAM

En colaboración con Naudit HPCN S.L.

Diego Sáinz de Medrano Otero
Captura y proceso de datos de aplicaciones, redes y sistemas a alta velocidad

Las siguientes tipografías se utilizaron en el documento

Noto Serif	Apache License 2.0
Merriwheather	SIL Open Font License
Inconsolata	SIL Open Font License

Para Natalia. Todos mis logros, notables o no, sin ti no hubieran sido posibles. ♥

Programming is not a science. Programming is a craft.
—Richard Stallman

Resumen

En el ámbito de redes de computadores, el mundo moderno se mide en gigabits por segundo, y no falta mucho tiempo para que la unidad de medida suba en un orden de magnitud. En negocios en los que la comunicación es un aspecto central, se convierte en necesidad implementar soluciones de análisis y estadísticas de tráfico por razones de seguridad y calidad de servicio. Este proyecto consiste en el diseño y desarrollo de un reemplazo para el stack tecnológico de procesamiento de tráfico en redes de alto volumen desarrollado por Naudit HPCN.

Se reemplazará un sistema de scripts en AWK coordinados por un script en Python por un único programa en Java usando la librería de procesamiento de streams Apache Flink. Además, se hará una mejora de la capa de almacenamiento de datos en PostgreSQL, tanto a nivel de inserción como a nivel de extracción, facilitando el desarrollo de la capa de visualización de datos usando Grafana.

Palabras clave — redes de computadores, alto tráfico, captura y procesamiento de datos

Abstract

In the context of computer networks, the modern world is measured in gigabits per second, and it won't be long until the unit of measurement goes up an order of magnitude. In businesses where communication is key, it becomes a necessity to implement network traffic analysis and statistics solutions for security and quality of service reasons. This project consists of the design and implementation of a replacement for the current technological stack of traffic data processing for high volume networks developed by Naudit HPCN.

A system of AWK scripts, cooordinated by a Python script, will be replaced by a single Java program using the Apache Flink stream processing library. Also, improvements will be made to the data storage layer in PostgreSQL, both for insertion and extraction of data, facilitating the development of the data visualization layer using Grafana.

Keywords — computer networks, high traffic, data capture and processing

Contents

List of Tables

List of Figures

List of Code Listings

1 Introduction

1. Rationale

One of the missions of the Naudit High Performance Computing and Networking company is to provide its clients with the tools and support to perform smart, fast and valuable analysis and diagnostics of their computer networks. With that objective in mind, they have developed a full solution for network traffic analysis which can be deployed in dedicated machines installed inside client's networks, which then run the software tools, including both in-house programs and open source tools.

This solution is composed of a mixed technological stack, performing the following tasks:

1) Live network traffic capture
2) IP conversation detection and statistics
3) Higher level data aggregation
4) Data rankings
5) Visual representation of the product data

This stack, further described in chapter 2, section 4, was designed for networks with moderately to high volumes of traffic, and is consistent of several "sections", implemented with different tools and programming languages, which are executed concurrently, producing a real time processing pipeline.

While the stack has been able to perform in a number of situations where traffic was moderately high, recent implementations with clients having massive volumes of traffic constantly have shown performance problems, especially in the middle section of the pipeline. However, attempting to improve and refactor the code involved has proven to be a difficult task. Some of the main problems are:

- The technology is very heterogeneous, involving a C++ preprocessor, a Python coordination script, AWK aggregation scripts, computer generated SQL functions to extract the aggregation results and compute the top ranking conversations.
- The codebase is fairly large and the different implementations distributed to clients have not always been kept track in a single version control system because they were developed in place, producing fragmentation.

As such, the goal of this project is to document the design and development of a replacement for the current tech stack. The fundamental objective is essentially "refactoring" the processing logic, keeping the fundamental tasks it performs intact while aiming for robustness, performance, data integrity and maintainability.

To do so, first we will tackle the classification and general documentation of the current programs and tools in use (chapter 2, section 4). Next, we will scout for a *stream processing* solution in which to implement the bulk of the work, looking for interesting APIs which are able to facilitate the data aggregation tasks, as well as having a nice interface with the storage layer (chapter 2, section 5).

With these tasks complete, and having selected our stream processing framework, we will then implement and test a stream processing job named **Blink**, and later refining its concurrency parameters as well as realising the possibility of implementing another of the described tasks, the data ranking section of the pipeline (chapter 3, sections 6 through 7.4). Afterwards, we will implement a data storage layer, aiming for a more robust storage, but also flexible and scalable data extraction capabilities (same chapter, section 7.5). Finally, we will produce a series of structured visualizations in Grafana (section 8).

Before starting, let's define some general concepts that we will be using throughout the project. While most of them will be familiar for the educated reader, they are exposed here for clarity, ease of reference and to give a more refined idea of the context and ambitions of the project.

2. Definitions

Let's begin with a series of incremental definitions of relatively general concepts that will be used througout the project. For any terms not included here, a glossary entry will be defined (B).

Conversation A subset of the packets passing through a network between two particular IPs, detected via identifying the origin and destination IPs.

Record A structured data object containing statistics about a given conversation in a span of time, like the number of connections realised between the endpoints, the average package RTT, the amount of bytes sent and received, etc. Records are identified directionally, that is, it matters whether the data flows from either endpoint of a conversation.

Aggregation Given a stream of records, the process of aggregagation digests the input, and "accumulates" the records belonging to either a conversation or a single endpoint (origin or destination), summing the statistical values associated with each incoming input. The choice of aggrupation, or the choice of the "key" of a record, separates the incoming records in a a key space.

For example, say three hypothetical records came in from the stream in a window of time:

	`srcIp`	`dstIp`	`numConnections`	`transferredBytes`
1	`145.233.144.66`	`8.8.8.8`	3	78
2	`145.233.144.66`	`92.180.109.242`	11	12567
3	`92.180.109.242`	`145.233.144.66`	6	99098319

Table 1.1: Record list example.

If the aggregation process is running and the key is set to be the `srcIp`, the columns of the records number 2 and 3 would be summed and accumulated in an internal state of the process, waiting for more matching records. Such internal state will also be called a record, specifying that it is an "aggregated" record.

Note as well that it is not necessarily the case that the columns have a numerical value, and it's possible to aggregate arbitrary data, given an implementation of the meaning of what the "sum" of any two objects are.

Ranking Given a batch (*not* a stream) of aggregated records belonging to multiple key values, a ranking process produces a similar batch, with the following modifications:

- For each field in the aggregated records, there will be a new column named `field_rank`.
- The value of the `*_rank` columns corresponds to the "ranking" of the corresponding record in the batch. Continuing above's example, where the two incoming aggregated records will have key values of `145.233.144.66` and `92.180.109.242`, the value of `numConnections_rank` for each aggregated record will be:

srcIp	numConnections	numConnections_rank
145.233.144.66	14	1
92.180.109.242	6	2

Table 1.2: Record ranking example.

Note that the values of the aggregations are still preserved in the new batch of what we will call "ranked" records.

3. Formal goals

Here we state the actual goals of the project in a concise manner. We will follow them as guidelines in the design and development process, and measure our success against them.

1) To substitute the current technological stack described at the beginning, from points 3 to 5 (inclusively), and obtaining a process equivalent in the resulting data,

2) to condense all the necessary code in a reasonable and documented structure, with a focus on code maintainability first, and efficiency second, and

3) to obtain a coherent project with easier operation, monitoring and adaptations than the original stack.

2
State of the Art

In this chapter we will discuss two main aspects of the state of the art technology: first, we will make a brief run down of the current technology stack implemented for the solutions offered by Naudit, and then we will explore the differences between some of the more popular modern stream processors available.

4. Naudit HPCN stack

Unless stated otherwise, the tools mentioned in this section are developed by Naudit in-house. In this chapter, the "client" is the generic organization which has hired Naudit for its network analysis services, and the networks will be the client's private networks.

4.1. Pipeline Setup

The typical setup for the provision of the Naudit services consist of a single installation of a dedicated server which is installed in the physical network of the client, where it can live capture live network traffic through dedicated 10Gbit interfaces. We will assume the capture and processing capabilities of the server, as well as the general OS systems, which all software running on the probe will use. This running environment will be a Linux based operating system, with a suite of custom suite of systemd service units made by hand to facilitate the operation of all the software.

4.2. Data ingestion

The starting point of the pipeline is the P2 tool. P2 is, essentially, an IP conversation aggregator. It operates in a minute-to-minute basis, processing the traffic in the network and generating "Records" based on the packet information. These records are represented as space-separated lines which are stored in timestamped files, which are produced at the end of the minute window of processing, and are generally stored in dedicated disks provisioned with RAID capabilities. The lifespan of this files is governed by file deleter scripts which work in conjunction with the rest of the software to maintain a fixed ratio of free disk space.

These files are the basic blocks of information that the following stages of the pipelines consume. Depending on the size and usage of the client's network, the typical size of each file is between 100 and 120 MB, which are very fast to read in modern hardware, so their production and consumption is generally not a bottleneck in the processing pipeline.

The P2 tool is not inside the scope of this project, and thus we will continue to be the foundation of the alternative solution that will be implemented. As such, further specification of what information the records store and how they are processed will be explained with more detail in chapter 3, but for now, let's stablish the basics:

Each space separated value in the record is referred to as a **field**. The number of fields per record is variable, depending on the configuration of P2, but for the particular use case concerning most implementations, including the one developed during this project, it will stay fixed. Each field can be *complex* or *simple*: complex fields will be parsed into dedicated data structures, and simple types will either be simple strings or numeric values. An example of a complex field is an IP address, and a simple field can be the layer 4 protocol of the processed packets, or the number of transmitted bytes, and most of the simple fields will be numeric values representing a statistic of the conversation.

Every P2 record represents a conversation between two IP addresses. Generally, conversations are a feature of layer 4 protocols, like TCP, but for the time being we will assume that a conversation mechanism exists, and the tool can detect packets flowing between two addresses and aggregate them into a single conversation. Conversations are limited to the minute-wide window of time in which the packets are processed.

As it represents a conversation, every record will contain two IP addresses, the one of the conversation starter, and that of the receiving end. This does not mean that all the packets in the conversation flowed from the origin address into the destination address: packets can flow in both directions, but it is the address which initiates the conversation which is represented as origin address. The rest of the fields will have a string representation of its value, or a default null value in case that particular field could not be computed for the conversation.

At this point we can mention a different data ingestion mechanism that is used in some of the client installations: NetFlow. NetFlow is a Cisco technology that can be used in a manner relatively similar to that of the usage of P2, in the abstract sense that it produces conversation based statistics, although its mechanism for doing so is quite different, as is its output format. However, once the NetFlow output is processed, the information obtained can be (and it is) used in very much the same manner as the information collected by P2, meaning that some of the implementations of the pipeline use it as a foundation.

At the beginning of the project, the possibility to replace both the ingestion points of P2 and NetFlow inputs was floated, but in the end the amount of work required to replace the rest of the pipeline took too much to include both. Thus, this is relegated to future work, but the design of the solution will take the different possible inputs into consideration.

4.3. Data preparation

In general, the data contained in the P2 records is useful enough as it is, but there are some commodities that are missing. As stated above, P2 processes conversations between IP addresses, but sometimes this is not enough, as the more useful information would be the domain names of the origin and destination points of the conversation. However, if we took the naive approach to resolve the names with DNS would prove too slow and unreliable: in some clients the range of distinct IP addresses is way too big to do this in a way that does not stress the system too much. Even if IPs could be resolved in a periodic way, for instance, once a day or week, the variation of addresses inside that time frame would make the resolution largely useless, as it would mean that to know the domains of many of the newly detected conversation endpoints would have to wait until the next name resolution, and blocking the processing until then is not an option, as the purpose of the project is to have as instant information as possible.

The chosen solution that works with most clients is to have a list of known IP subnets which can be related to a single name. As big organizations grow their networks, there is a tendency to name the different subnets inside of it so that there is a more human approach to classify and refer to different sites, whether those sites are physical or simply abstract separations. This is the use case that will be considered for this project.

The concrete solution currently in use consists of two parts:

- A file acting as a database of the many-to-one relation between different subnets and sites. It is a CSV file that is updated by the client as they see fit.

- A program that acts as a preprocessor for the data that the pipeline will consume, written in C++, which parses the relevant IP addresses and matches them with their corresponding subnets.

The input of said program is the initial data files produced by P2, split up into single records, and the output is the stream of records, modified with a tag for "source" and "destination" sites. This stream is consumed by the next phase of the pipeline.

4.4. Data processing: aggregation and rankings

Once the records are prepared, they are ingested by a series of different AWK scripts that perform aggregation based on different selected fields. The way they aggregate data from the incoming stream of records is to maintain a hash table with different keys, which are a specific field of the record, containing a set of the values that are to be aggregated. Then, for each record that comes in, the key is matched, and the values stored in that record are accumulated into the corresponding entry in the hash, which generally means that the values are summed. There is another kind of script that, given the output of an aggregation, it produces a ranking of the top records in the batch. These rankings are performed for a number of fields of the records, meaning that for each such field, a record is given a ranking number, and in that way, conceptually at least, several rankings are produced, one for each field desired.

These scripts are parametrized with the amount of time they must spend processing the input, and several instances of each script are launched with incremental values for said duration. At the end of each window of time, the aggregated records (that is, the entries in the stored hash for the window) are output and then consumed by the next instance of the same script–for example, if two sets of scripts are launched, one with a window of 1 minute and one with a window of 5, every minute the aggregated values from the first script are sent as input to the second one, and the shorter script is reset, starting a new aggregation from zero values which will incorporate the information of the next records. The same thing happens with every stage: at the end of their specified time, their accumulated records are output, and if there is a next stage, it consumes them as input.

The way this mechanism is deployed is via a single Python script. A configuration file with the desired intervals of aggregation is read, and then the script deploys the corresponding number of instances of each kind of script. This deployment is done via OS threads, so that the script can redirect each script's standard input and output, and that way the records can be sent to their corresponding destinations. This is done using queues, which connect one script's output with the next's input, but this is not their only use: every record that passes through a queue is then stored in a temporary file, which acts as short term storage before the data is sent to a persistent storage layer (the database described in the next section).

While the aggregation scripts themselves are not awfully complicated, they are rather inaccessible to new programmers that come accross them, having been written by people very familiar with the records' structure. Besides, every time a new field is needed in the aggregation, or some of the information that needs to be aggregated requires some computation (like parsing a complex field to get some values), this change needs to be implemented in each different kind of script, and the AWK code needed for some of these changes tends to be rather complicated. There is also a complexity problem in the central Python script, which in effect has grown to be an entire deployment system of its own. The code manages the data ingestion and preparation, as well as the thread launching, the coordination between scripts and the later storage of the aggregated data.

4.5. Storage and visualization

After each defined window of aggregation time, the aggregated records are passed down to the next stage of aggregation, if there is one, and it is also sent to a PostgreSQL database. The database is fitted with a time series extension, TimescaleDB, which helps with performance when dealing with time series data. For storage, a series of tables are defined in the database and are marked as a "hypertable", a TimescaleDB concept that indicates that a table can be "partitioned" over time. This partitioning of

the tables provides absolutely transparent optimizations to the storage and extraction processes when they are done over a single partition or “chunk”, which in turn represents a time interval.

For every level of aggregation time and script kind there is a corresponding table in the database, and it is partitioned in accordance to the interval of aggregation: for example, if an aggregation process is done in a span of 5 minutes, the corresponding table is partitioned hourly, and if an aggregation is done hourly, its corresponding table is partitioned daily. This way, we ensure that batches of data remain in relative proximity in time after being processed in the pipeline, which makes for faster querying later on.

Aggregation span	Table partition
1 minute	1 hour
5 minute	6 hour
1 hour	1 day
1 day	1 week

Table 2.1: Example of table partitioning schemas.

This mechanism is working as fine as it can be, relying on state of the art database technology, and so it will remain outside the scope of this work, safe for minor adjustments to table columns, names and possibly partitioning choices.

For the visualization layer, which relies heavily on the storage, the most useful and logical tool to use is Grafana, a very powerful dashboard platform. With it, a series of dashboards are laid out, each including different panels displaying graphics, which in turn are plotted using the real-time data coming from the storage layer. Grafana provides very well adapted interfaces to work with many datasources, including PostgreSQL instances, which facilitates a lot the extraction and plotting of time series data.

Figure 2.1: Example of a generic Grafana dashboard.

Grafana, being centered in time series data visualization, provides the user a time range selector, which can in turn be utilized by the datasources in order to request information in that specific time range. In the case of PostgreSQL, this is done by providing macros which can be used in the SQL queries to the database, which are translated in real time to convert to timestamp values, and then the queries are executed against the selected PostgreSQL instance. As will be described in more detail in the design and development chapter, a mechanism has been implemented in the PL/pgSQL language to perform a dynamic selection of tables to look up information according to the selected time range, keeping a semi-consistent number of datapoints independently, which not only helps with the consistency of the presented visualization, it is also helpful with the performance of the pipeline: attempting to select every datapoint for every minute in the last 7 days can prove to be too much for the browser, where

Grafana is running, even with maxed out desktops, so it is a better tradeoff to present a less dense aggregation instead.

The dashboards deployed in the current implementations are working basically as intended, similarly to the storage layer, and the goal is to replicate them with high fidelity in the finished product, if not exactly, and at least representing the same basic information.

While there are a lot of dashboards, only a limited number of them will be included in this document, as time constraints put this part of the development very far out into the course of the project. We will consider three of the more important dashboards:

- one that includes graphic information about the top ranking sites by the different aggregation fields,
- and two which display specific metrics for selected sites in the sample space, overlapping in the same time interval.

As the replication of these dashboards will involve interacting with the storage layer in very intricate ways, the details will be explained further in the relevant section of the next chapter.

With the rundown of the most important key points of the current processing pipeline out of the way, we turn to an exploration of the possible technologies that we may use to solve the performance and organization problems while accomplishing the same goals.

5. Stream and batch processors

When a constant input of new information is presented, which has a potential to be extracted and processed in order to obtain valuable data, **batch** and **stream** processors come to the development process. These terms encompass many systems, but the gist is this: a batch or stream processor is a system which takes a potentially infinite inflow of data, transforms it in some way that makes the information more useful or concise, and then produces a simplified output interface for the transformed data. Batch processors are those that stall for some amount of time to accumulate some of the input, and then process the buffer (or batch) before continuing, while stream processors do away with the batches and provide abstractions atop the data stream itself.

Traditionally, big data processing jobs have fallen into the “analytical processing” category, and have been performed in dedicated clusters using the ETL paradigm:

> An ETL process **e**xtracts data from a transactional database, **t**ransforms it into a common representation [...], and finally **l**oads it into the analytical database. (Stream Processing with Apache Flink [KH19])

The data processed in such processes can come from many different places, such as bussiness applications metrics, ERP systems, web applications, etc., and be very heterogeneous in both volume and format. It is the job of the different ETL processes to collect such data from its sources, transform it so that it becomes more usable, and also coordinate with other ETL processes to maintain a synchronized data warehouse, which can then be queried to produce useful information. These processes can become very complex and become hard to maintain very rapidly, even though very powerful tools from the Apache distributed ecosystem have been available to help build them (like the distributed filesystem HDFS, or the SQL-on-Hadoop engines Apache Hive and Apache Impala).

The rise of stream processing technology began at around 2011, where a first generation of stream processors patterns emerged. They consisted of secondary layers to traditional batch processing layers and were implemented to optimize for speed instead of accuracy: by doing this, more inmediate answers could be extracted from the data sources, and then be corroborated by the exact results computed by the traditional pipeline, even if it meant trading off the precision of the results.

The second generation, circa 2013, aimed to improve the precision and failure guarantees provided by the newer pipelines, as well as to expose higher level APIs as data operators which could work with a higher throughput. The tradeoff was an increase in latencies, which had been the major improvement of the previous state of the art solutions, and a higher dependency on the performance of the ingestion of events.

We will focus in the processors of the third generation, which emerged around 2015. They are characterized by the improvements made towards the erasure of the throughput vs. latency tradeoffs, as well as the generalization of the "exactly once" semantics, which are the mechanism by which the processors guarantee accurate results even in events of failure. These features meant the obsolence of the previous generations, as it bridged the latency gap while maintaining the traditional batch processing precission.

Usually, stream processors are conceived to run in computer clusters. This is because the volume of the incoming data is typically humongous (Twitter or Google analytics, big sites traffic information, etc.), but many of them usually offer single machine solutions as well, for smaller streaming jobs. In the search for good matches for our use case, we have reduced the selection to the following technologies, all coming from the Apache ecosystem:

- Apache Spark. One of the first platforms that come to mind when exploring solutions for Big Data and data analytics. In particular, Spark Streaming is the extension of Spark that seems to fit the bill the best: it is the API (with bindings in Python, Java and Scala) that provides with the operators and utilities to perform transformations of the incoming data, and those operations are then run on the Spark engine. Pros of using this solution is the vast amount of community support for Spark, and a tight integration with Spark and other Apache systems, like Kafka, HDFS, etc. However, the APIs are in general oriented towards *stateless* stream processing, centered in a functional paradigm, and it becomes complicated to attempt to produce stateful processing and coordinate the different processes to maintain state across an entire processing pipeline.

- Apache Storm. Another stream processing system. It's main difference from Spark is that it is its own platform, rather than being a layer atop a distributed computation engine. It is stateless by default, but it provides APIs for stateful management; however, it is an opt-in feature, rather than the default. There is also a layer for achieving exactly-once semantics, called Trident, but again, this is an opt-in addition. Some of the best pros of using Storm for our objective is the ease of deployment of computation clusters, and it also shares a nice integration with the rest of the Apache ecosystem. It also has a very interesting capability to run its "bolts" (the processing units of the pipeline) defined in other non-JVM languages [Fou19b], like Ruby and Python. However, as mentioned before, the stateless nature of the processing makes it difficult to fit our needs.

- Apache Kafka. Not exactly a stream processor, but its characteristics as a distributed message system makes it worth our attention, because it could be used to extend even further the exactly once guarantees of our pipeline. As we are capturing live traffic, for some intervals of time downtime could mean data loss. This could be remedied by running a Kafka instance in that point in the pipeline, providing assurance that all the file creation events coming from P2 would be queued and then processed fown the line. However, while this would be ideal, considerations of memory in the dedicated server, which would be running a lot of services besides our pipeline, lead us to the conclussion that the tradeoff is unnnecessary, as we should focus in the extremely high proportion of the time where the pipeline is running with no problems, and not in the very little possible downtime, at least in a first iteration of the development. The relative ease to integrate it with the rest of our selection of Apache systems would make it very fast to integrate, though.

In the end, however, we chose **Apache Flink**. Flink is a similar stream processor as those we just mentioned, with a key difference: it is *stateful* by default. This means that when we set up our processing pipeline, we can take advantage of the stateful nature of the computations inmediately, without

having to set it up (but we can tweak the state mechanisms should we need to). In addition, Flink exposes a very full-featured API of Functions, Flink's denomination for the operators of the pipeline, including classes we can use as a base for our operators like `AggregateFunction` and `ProcessFunction`, and data extraction and storage classes, such as `SourceFunction` or `GenericWriteAheadSink` (which you can read about in more detail in the Flink website [Fou19a]). Flink exposes it's functionality in a very expressive way, in which we can define the main lines of our processing pipelines in just a few lines of code, separating nicely our data representations and the processing logic. We can consult an example of a program that listens on a socket for words and counts the number of different words sent every 5 seconds in the example listing A.1.

In the example we can also see that Flink provides common aggregation techniques, such as summing tuples over some of their fields, as standard parts of its library, but we will rarely use them for our case. They are, though a nice bonus to help the process of understanding the programming flow of Flink.

For further reference in the use of the Apache Flink system, we rely heavily in the *Streaming with Flink* [KH19] book. It lays a lot of the foundations needed to implement simple pipelines, knowledge that we can use to extend the functionalities and the scaling of the system.

To summarize this section, here's a small table in which some comparisons are made between the different contemplated options.

	Spark Streaming	**Storm**	**Flink**
Stateful	Opt-in (`updateStateByKey` function)	Opt-in via extra library layer	Yes
Standalone runtime	Runs as a Spark layer	Yes	Yes
Language bindings	Python, Java and Scala	JVM, and non-JVM through a DSL layer	Java and Scala

Table 2.2: Stream processors comparative table

5.1. Extra features of Apache Flink

Flink has a series of semantics and mechanisms that aid developers in the production of stream processing jobs with extra guarantees and generally nice things to have.

For one, Flink lets the programmers choose what kind of timestamping they want to have on the records that flow through the system. There is a main choice between 'ProcessingTime' or 'EventTime': the former is simply the time that the record was first perceived by the process, and the latter is a more sophisticated stamp that can be provided by a certain logic. We will use P2's output format, described more in detail later, to determine that the event happened when P2 finished processing, which is a bit earlier than when we are done processing it through our first stages of the pipeline.

There are a number of other features that we will make use of in the development of this project. To name a few, described in the introductory chapter of the reference text [KH19]:

- Highly optimized runtime: "Millisecond latencies while processing millions of events per second. Flink applications can be scaled to run on thousands of cores."
- Various levels of control over the data flow, represented in the so called "layered APIs", which allows for higher and lower level operations.
- A very pleasant collection of automatically added metrics that aid to monitor the performance of the jobs and the distributed system, such as Prometheus metrics.
- A web app that runs in the background and provides operation access and real time display of the running jobs.

3
Design and Development

And so we begin the development of the replacement pipeline, which will be referred to as **Blink**, its project codename. As stated in the previous chapters, for our project we will be replacement only certain segments of the pipeline, where we can finde the most acute performance and maintenability problems. So, in the following sections in this chapter we will be describing the development process of , including the design phases and some interesting observations made through trial and error between design iterations. Before that, we will provide a better introduction to Apache Flink's programming paradigm of streaming jobs than the one made in the previous chapter (in section 5).

Throughout the chapter we will see some code repetition patterns in different parts of the implementation. This lead us to consider a system for code production automation through templating engines, with what we call code "recipes". This concept became a very big consideration while the project was pretty advanced, as we started to appreciate the capabilities that such a system would allow, but even though some plans and initial prototypes where made, due to time and project (meaning *this* project) size constraints, it will be relegated to the future work category (see chapter 5, section 12).

6. Flink Streaming Jobs

6.1. Introduction

Flink is a stream processing system which provides a very nice programming interface to write data processing jobs, or **streaming jobs**. But let's define what such a job is more specifically.

Streaming jobs can be represented as directed graphs, where the nodes are classified in three categories: sources, functions and sinks. Except sinks, the nodes are defined by Functions in the Flink nomenclature—this exception makes sense, as sinks are not meant to produce outputs. A streaming job is defined by what its data sources are (which source functions will be used as data intake), what operations will be made on them (the generic function nodes) and where the operation results will be stored (what sinks will persist the final output). Another layer on top of this graph is the one where the time managment is described. Atop this operator functions we can specify time windows of different types, which determine how the operator will behave over time. These windows are usually used to partition the input event stream, so that only events belonging in the same window are processed together in the same operator. They are defined seamlessly with the rest of the pipeline definitions, as will be shown when we get to some code examples, but are harder to represent in the 2D graph image.

Let's look at an example from the reference book [KH19, ch. 2], which lays out in a very abstract fashion what the pipeline is supposed to do:

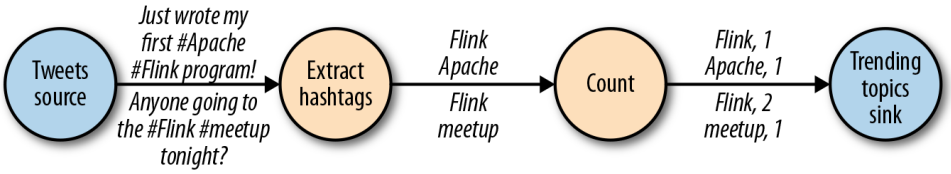

Figure 3.1: Example of the logical concept of a streaming job concept.

When introducing stream processors we said how they were generally designed to be run in clusters. The way Flink implements this feature is by implementing "task parallelism"–when it is detected that multiple instances of an operator can be safely run in parallel, Flink creates multiple operator tasks which will run it, and connect it automatically to the rest of the related logical nodes. These connections are managed by the engine, and if the "task manager" process determines that nodes lay in different machines, they will be implemented as network connections transparently. It is also important to consider the matter of how the data is passed through this new graph, and this is referred to as data parallelism strategy. This strategy determines how output from a node will be directed to the instances of the parallel node inmediately following the sequence. Although there are a lot of possibilities, through the project we will recurrently use three of these strategies:

Forward When there are no parallel operators involved, or there is no need to apply special rules over the data redirection, this is the strategy used by default. It has the advantage that, if both operator tasks are in the same machine, they are suitable for "task chaining", which is an optimization over the regular connection mechanism (as there is a lot to say about this mechanism, we will leave that to the reference book [KH19, ch. 3]).

Broadcast This strategy consists of the replication of all outgoing data to all connected operator tasks. It is used when the same output must be used in different tasks—for example, in our case, when the result of a window of aggregation is output, it must be passed to the next aggregation phase as well as to the corresponding storer. As this strategy involves data duplication, and possibly network connections through which to send said data, it is an expensive one to be used with care.

Key-based This will be the most common strategy in general, as in general it is the most useful. It consists on extracting a key from the events passing through, which then will be used to determine exactly to which operator task it will be sent. For example, one may use a tag in the events, and partition over the key space of different tags[1].

Note that these strategies are not generally selected explicitly (with the exception of key-based), but are rather implied from the definition of the linking between the operators in the streaming job. When Flink lays out the "real" job, it selects the correct strategies transparently, but it is a good idea to know the different strategies well to make better choices when defining the streaming job.

6.2. Blink streaming job summary

With the introduction out of the way, we can now produce a diagram showing in a very simplified way the layout of the job that we will be implementing in the rest of the chapter.

Our running environment will be a single server running a Linux based OS, rather than a distributed system, but that distinction will be easy to address if in the future the need arises. This machine has 48 virtual CPU cores available, as well as 200G of memory. We will configure Flink, however, to only consume a maximum of 48G.

[1]This strategy does *not* imply that Flink will deploy the same number of operators as different keys in the key space. Parallel operator tasks can share the same key space and be assigned an even distribution of the incoming keys dynamically, and maintain them separate without further setup. Again, the reference goes way deeper into this mechanism.

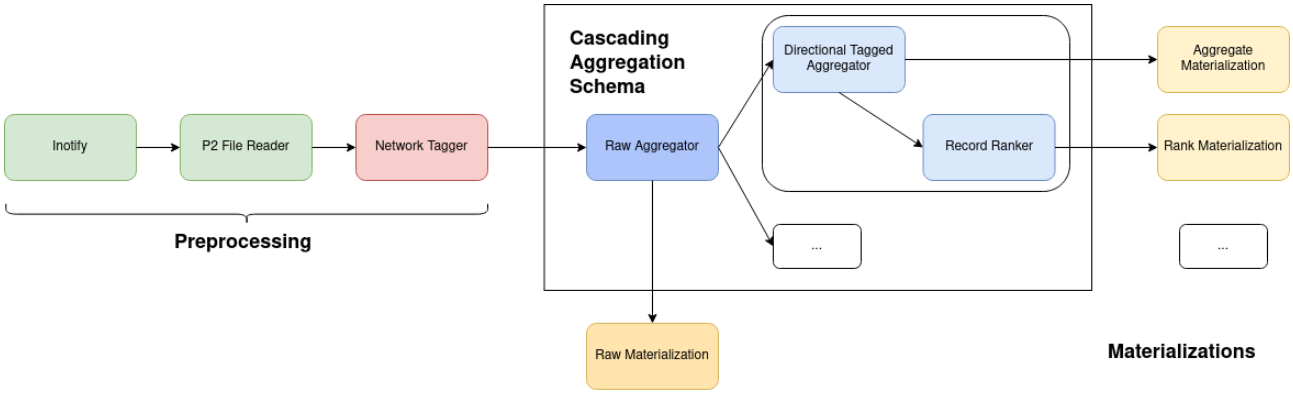

Figure 3.2: Abstract dataflow of the Blink processing pipeline.

All extra information needed by the operators defined in the pipeline will be provided in a simple configuration file written in YAML. This includes a development switch, which is intended to switch on latency metrics across the pipeline. We will use some of this metrics for performance monitoring in chapter 4, albeit in a limited manner. What will be used are the official Prometheus metrics integration, which is activated by simply dropping a JAR file in a specific directory, and then we benefit from many useful metrics about both the running job and the general Flink system.

As mentioned before, in describing some of Flink's capabilities, we will be using a custom EventTime for the records that the pipeline will process. We will also use the same timestamps plus a minute for the generation of "watermarks" and "checkpoints", two Flink related systems that are used to provoke savepoints in the job execution and coordinate state saves. The setup consists of checkpoints delivered every minute, which gives us a strong guarantee that Flink will be able to restore the jobs in short order and not have many data losses in the process.

7. Blink

7.1. Data ingestion

As stated previously, we will be working with the P2 tool as a source of information. The way it produces its data is as follows:

1. It begins its capture and processing of network packets.
2. When a minute passes, it creates a series of files, timestamped with the corresponding Unix epoch, and begins dumping the processed output into it.
3. When the dumping is finished for each file, it produces a write and close event to the filesystem, and imposes no further actions on them.

The filesystem events produced in steps 2 and 3 are the only means available to detect progress from the P2 tool. The way we will listen to such events is through Inotify, a Linux kernel utility that extends filesystem capabilities and provides with programs that can listen to them. This means that we lock our process to run only in Inotify compatible platforms, which are essentially the Linux ecosystem, but since the majority of the low-level implementations of the rest of the Naudit tools, including P2, require a Linux environment, this will not play too much to our disadvantage. Should the need come to extend our platform support, for example if we are to extend the range of machines where we will deploy our clusters, this is the only part of the pipeline that is locking, so in theory we could tweak the deployment strategy to put the corresponding source interfacing with Inotify in the same machine running Linux and P2, and leaving the rest up to the distributed engine.

While there are libraries to interface with Inotify, the explored Java bindings that we found were not mature enough, or lacking some functionality. Instead, what we will do is use one of the Inotify

tools, `inotifywait`, a tool for gathering filesystem statistics. These are two of the flags (included in the Debian distribution) we will use with it:

- `-e <event>`: the event parameter is a string which must belong to a set of names, each describing a filesystem event. We will be listening only to the `close_write` event, but the flag can be specified more than once to add other watches.
- `-m`: this tells the program to continue running after detecting an event, instead of the default behavior, which is to exit.

After that, we will pass it the directory where P2 dumps its output, and all filesystem events ocurring under that node will be processed by the program. Here is an example of the program in question in use, with additional flags for easier formatting:

```
$ inotifywait -e close_write -m --timefmt "%s" --format "%T -> %f"
Setting up watches.
Watches established.
1592512681 -> _tcp_@1592512677
1592512681 -> _udp_@1592512677
1592512681 -> _other_@1592512677
1592512681 -> _eth_@1592512677
1592512681 -> _arp_@1592512677
1592512681 -> __@1592512677.finished
...
```

We can see the format of the P2 files laid out. The files we are interested in for this project are those that match the following regular expression:

```
^_(tcp|udp|other|eth|arp)_@([0-9]+)$
```
(3.1)

1. A first field, with the type of file surrounded by underscores,
2. and a second field after the at symbol with the Unix epoch.

This criterion leaves out other possible P2 outputs, which are not relevant to the project.

Note as well that the time of the event and the file produced in that event do not match in the provided output. This is because P2 tags the file with the timestamp of the beginning of the window, but finishes some time afterwards, depending on the volume of traffic processed.

So, onto the implementation of our entry point of the pipeline. We will use a class that implements the `SourceFunction` interface. As per the documentation [Fou19a], we must parametrize the class with the desired output type. In our case, we will be producing three values per event:

- a `Long` value with the timestamp (in ms) of the file (which will also be the event time of the produced object),
- a `String` value with the complete path of the file, which will later be used to read it,
- and another `String` value with the type of file, which, albeit not exactly precise, we will denote as the protocol string.

In order to listen to the events, we will launch an instance of the `inotifywait` program as a subprocess with the "`-e CLOSE_WRITE -m /path/to/p2/output`" arguments, which makes it produce in the default format:

```
<full path to the directory> <comma separated events> <filename>
/path/to/p2/output/ CLOSE_WRITE,CLOSE _arp_@1592515197
```

The directory to watch with `inotofywait` will be passed down as a parameter to the class, and will be obtained from the global configuration file.

After launching the subprocess, we will launch a loop which will continuously read the output of the program, separate them by line carriages (as sometimes the standard output reading produces multiple lines in one read), and capture the desired values from the space separated columns. After that, using the regular expression 3.1 we will extract the protocol string and the timestamp, and produce an event of the `Tuple3` type (a Flink utility class), with the three parameters specified before, and marking its event time as the extracted timestamp (multiplied by a thousand to express it in ms).

While this is good enough for some applications, in this case we want to have some more guarantees of processing. If we left the development at that, any failure in the connection between our source operator and the rest of the pipeline would result in the loss of any missed filesystem events that are not transmitted to the file collecting unit (detailed in section 7.2). For that, our class will also implement the `CheckpointedFunction` interface, which allows us to implement more sophisticated state mechanisms. In the interface methods, we will launch the subprocess and tie its output to a queue. For the state, we will store every line of output in the checkpointed state and drain from it as items are processed off the queue. In that way, if and when crashes occur, the state may be recovered from the last checkpoint and fed into the queue, thus producing events from the missing runtime.

This is not a bulletproof system, though, as the death of the `inotifywait` subprocess may produce a small window of time before checkpoint recovery where there was no event listener, and some files may be produced by P2 in that window. In this scenario, those files would be missed by our source function, and thus not be processed later. However, the robustness of the queue mechanism is enough for a decent prototype, and further developments in this regard can be left to the future work category (chapter 4).

7.2. Preprocessing

We now enter what we will call the “preprocessing” part of the pipeline. Here we define the operators which will take the input events (that is, the previously described `Tuple3`s) and produce the complete Record stream, ready to be aggregated in the cascade (next section 7.3).

First of all, though, we must get at the actual data. Although our model 3.2 states that the source of the dataflow is the P2 records, technically for our pipeline the input objects are the P2 “file specifications” described in the previous section. With those specifications we will be able to read the files and produced well formed objects that we can make use of. Thus, to the end of the `InotifySourceFunction` we connect the following chain of operators.

File collecting. The first step is to read the files and extract it in a useful format. To do that, we will use the second item in the received `Tuple3`, which stores the path to the file to be read. A standard file-read-loop that goes through the file line by line suffices to do this, and then we split each line into its corresponding fields by the empty space separator, “ ”.

The Flink API provides a perfect match for this kind of function: the `FlatMapFunction`. The `flatMap` method a class implementing this interface must have takes a single input but may produce any variable number of outputs. That is exactly what we do: for a single file, we will produce many line objects, specifically an array of strings. Our model for such array will actually be a Flink’s `Row`, a variable lenght object that provides with nice accessor methods. That, together with the extra information that was passed into the function originally, will be the actual output. Thus, we implement the `FileCollector` class as a `FlatMapFunction` parametrized to output a very similar to that of the input: a `Tuple3` with the timestamp, the row and the protocol string.

Typing. Here is where the extra information we have been passing down the pipeline will be merged with the actual P2 data to form a class that will be the ‘true’ data representation at the be-

ginning of the core of the aggregation process: the Record. The idea is to use the protocol string to determine the exact specification that the fields, represented now as the row's fields, and store them in a class as members. If we were using a language with data structure support, or even a JSON representation of the data, those alternatives could be used, but in the Java world we must implement a class to represent the same data and have named members instead of row numbers for access to the fields.

This is not all bad, however, as this is the perfect place to implement the complex field parsing[2]. If we remember when whe talked about the different P2 fields in section 4, these complex fields were essentially strings with a specific format to encode more complicated information than a scalar or simple value. For an example of this, there is an RTT field which contains the following information about a single record, that is, referring to a single conversation analyzed by P2:

- the number of samples taken to measure the packets RTT,
- the minimum RTT perceived during the conversation,
- the average RTT perceived during the conversation and
- the maximum RTT perceived during the conversation.

In addition, more than one of these samplings may have been applied during the processing, and each sampling is encoded in the same field, separated by a special character.

So, the implementation of the Record class is as follows:

- There is a constructor which receives the information produced by the file collector function. It switches on the protocol string and takes the relevant fields from the row through their index: the field-to-index relation is obtained from P2's documentation. For this project, three protocols were implemented (TCP, UDP and OTHER), as they are the most used for the client sites, and they contain many overlapping fields, which makes them easier to process in the same pipeline.
- It implements any *public* subclasses to represent any complex fields and *private* methods to parse them from string fields as necessary.

Here we made a design decision related to the language: all of the class members are made `public` and those containing data are editable (that is, not `final`). This makes accessing the fields of records *much* more simple and robust than the traditional getter and setter schemas dogmatic to the Java and friends class of languages. If this was not enough justification (which as a developer it already is), some number of reasons could also be provided:

- this development is intended to be a closed system where only classes written for it will ever access the records,
- there are no intended side effects to the accessing of these values: the volume of operations performed on them is to be so large that attempting to implement additional behavior would not be worth it,
- the only reason we have members instead of a different system is our necessity to have heterogeneous types for the fields and Java does not provide sufficient support for data structures, and having direct access to the members is the closest thing to a hash table or JSON approach.

Together with this class, to the end of the file collector function we connect a `MapFunction`: a one-to-one converter that takes the tuples and returns Records, which are almost ready to use.

Network tagging. As mentioned in the previous chapter, traffic as is processed by P2 is not inmediately usable, as it distinguishes conversations based on IP addresses, which are not always resolvable

[2] As well as other operations not discussed in this section, but that are very useful: for example, the NetFlow data can contain fields which represent sampling scales of other fields, which we could transform by mutiplication to maintain a single scale for the rest of the pipeline.

to domains, or may not even have an associated domain. However, there is a book-keeping file where the different subnets, written in CIDR format are tagged with additional information. Each subnet is tagged with two fields: the site to which the subnet belongs, and the "service" that said site is dedicated too. This is just another subclassification for the subnets, as per-site classification is not always useful enough to determine network problems.

The relation specified in this "database" is one subnet to one site and service pair—a site may appear as the tag of multiple subnets, but each subnet will have a service to distinguish it, and the same with the services.

What we will do is introduce this information into the record class so that it is usable for hashing, taking one of such tags as keys, down the pipeline. We accomodate this by adding extra memebers to the Record class, in our case

- `dstNet` and `srcNet`
- `dstSite` and `srcSite`
- `dstService` and `srcService`

In order to add these tags, however, we need to use a mechanism to translate IP addresses to the subnets they belong to, given a set of subnets. LPM is an algorithm that performs such classification. It is based on the use of a trie containing suitably ordered nodes against which we can match IPs and compare which node has the longest match from the beginning of the IP address, hence the name of the algorithm. There is a Java implementation of this algorithm by a class which models a trie, implemented inside Naudit, so we will not discuss it further than it's API: it is a parametrized class, meaning we can assign any type to be the value stored at the nodes, and it has an `insert` method which can insert subnet-value pairs maintaining in-trie order, and a `lookup` method which retrieves the value stored at the best match. There is a guaranteed default subnet with the null values against which any IP will match, in case a looked up IP matches no subnets in the trie.

So we implement a class named `NetworkTagger`, implementing again the `MapFunction`, except this time we will leverage Flink's layered APIs and use a more low level, more complex class: the `RichMapFunction`. By implementing this interface, not only do we need to implement the `map` method (which will take Records to Records), but also requires the implementation of the `open` method, which is called at the beginning of the lifecycle of the operator, and we will use it to load the subnets file into an in-memory trie. At the time of development, this file was a CSV file with three columns, one for the subnet, one for the site name and one for the service name. Using a simple CSV reader does the job well enough. However, there are future developments which will translate this data to a relational database, because inconsistencies and maintainability issues have arisen frequently enough with the keeping-up with the file. Switching the reading mechanism should not be very difficult, though, as it is simply a matter of implementing a simple SQL selection statement.

Once the trie is filled, the function is operational, and any Records coming through it will be output at the end, having the tags filled (if there are matches for their corresponding destination and source IP addresses). If there are no matches in the trie, the subnet tags of the record are defaulted to the IP. We will revisit this defaulting in the section about the aggregations.

And with that last step, the Records are now prepared to be processed by the main core logic of the aggregation scheme described in the following section.

In a way, we could regard the Inotify source function, togethter with the preprocessing segment described in this section as an "adapter" if we consider an hexagonal architecture approach [Coc05], and our core domain would be the actual aggregation processes. With that lens, we could regard the Blink data inputs as the preprocessed Records. If in the future the project gets extended and a new datasource requirement appears, this division of concerns will be very useful, as we could simply create new adapter segments for those datasources, modelling them with a new Record class, and still perform the same core logic. In fact, this is precisely one of the great attractiveness that switching the technology

stack to Flink provided, because once a datasource is integrated, it becomes easier to implement new datasources.

7.3. Cascading aggregation

The segment described here is the main focus of the Blink project, the aggregation scheme. Seeing the data input as a stream of Records, this scheme processes it to extrat time series data at different time windows containing traffic statistics in the tag level instead of the IP conversation level.

The whole aggregation process will pivot heavily in the model class of Record, or rather, the next iteration of them. As it is implemented up to this point in the development, the Record class is merely a data structure with a constructor and constructor helper methods, useful to work with column names instead of column numbers. For the initial development process it was decided that this class will remain as it is, and not be extended further, and instead we will implement a new class, the *AggregatedRecord*, which supplants the Record as the basic unit of data down the pipeline. This is done for the reason that the Record takes its member names directly from the P2 column specification, and thus is a closer representation of the original data. This is good in a certain sense, but since the development atop P2 does not necessarily need to follow the same conventions, it would be preferrable to have more "appropiate" or at least easier to interpret field names.

With that in mind, we find it better to implement a new class with such names, which is adapted to "absorb" the previous Record class, and with very useful methods for the aggregation process, rather than refactor the previous class. This is not the most efficient way to do things, at least in the development sense, as we keep multiple representations and that could lead to some confusion. However, merging these two approaches, and having a single representation coming from the preprocessing adapters and with the useful methods we will proceed to describe, will be relegated to the future work category.

Aggregators

For the aggregation process we will leverage Flink's API by implementing the `AggregateFunction` and using it as our main operator. The described Records (initially) and AggregatedRecords will flow through the instances we will spawn of this operator. Flink will use our class implementation to perform *rolling aggregations* in the defined time windows of our choosing; for the lifespan of the time window, records flowing in will arrive and be accumulated onto a single AggregatedRecord (Records will be "accumulated" into AggregatedRecords through a method described in the next subsection, and the same with AggregatedRecords accumulated against each other), and once time is up, it will output the result of aggregation as the accumulator value.

We will implement two such functions: one for the "raw" aggregation at the beginning of the pipeline, which will consume Records and will aggregate based on *pairs of sites* in a minute window basis, and one which will consume AggregatedRecords, which will aggregate based on the direction of the conversation and will have an aggregation key corresponding to one of the tags of the records. The former will be deployed only once, and the latter multiple times, and the number of operators will be (not taking into account the parallelism of task operators):

$$(\text{two ways}^{1}) \times (\text{number of kinds}^{2} \text{ of aggregation}) \times (\text{number of time levels of aggregation})$$

where we have the concepts

1. **way** is one of two values, from origin to destination or the reverse. It is modelled with an enumeration with the values `SRC` and `DST`.

2. **kind** refers to the tag that will be used as key in the aggregation. It is modelled with an enumeration with the values `SITE` and `SERVICE`.

For the normal functioning of the end result, all options will be selected for every time window, but for testing or measurement purposes we can reduce the number of operators by choosing to aggregate only per service, for example, or only in one way.

The reason for the division between raw and normal aggregation is the following: the goal of raw aggregation is to translate IP level conversation statistics produced minutely by P2, to subnet-to-subnet level conversation statistics. By matching the interval we will be able to translate a single P2 output to a single batch of minutely subnet-to-subnet conversations bundle, which will already be a reduced size stream, and upon that aggregation we can build the rest of aggregations, which we will want to be more specialized to aggregate on single tags and directions.

Thus, every aggregation operator of the second kind will be connected through the pipeline to the output of the first raw aggregator. In a following subsection we will talk about how the operators are actually deployed.

With all this, we implement two classes implementing the `AggregateFunction` interface: `RawAggregator`, parametrized with Records as inputs and AggregatedRecords as both the accumulator and output, and `Aggregator`, with all parameters set to AggregatedRecords. This means that both functions will generate an accumulator value at the beginning of an aggregation window, that is, an instance of AggregatedRecord, and it will accumulate incoming records into it, and at the end of the time window they will output said accumulator.

To accumulate such objects, the Aggregator classes implement the "add" and "merge" methods, as required by the interface: however, for this case we choose to make these implementations actually trivial. The simplified code example is listed in the appendices, A.2.

The reason to do it in this way is that the project is very sensitive to requirements changes, and implementing the necessary aggregation logic in these classes would mean that if the base class representing the data changes, then we would have to refactor code in multiple files, while this approach allows us to use the object oriented nature of the language to keep the data and operations pertaining to it close. For instance, say that we implement the previously mentioned Netflow adapter and must change the Record and AggregatedRecord classes to accomodate for the different fields—in this situation, by delegating the logic to the Record classes, we can reuse the Aggregators without changing a single line of their code. This also means that if we have described an aggregation schema, with the relevant time windows defined and the whole layout described, *we can also reuse it*.

AggregatedRecord

But in order to have such advantages, we must first implement the adding / merging logic in what will be the accumulator throughout the whole pipeline, the AggregatedRecord.

The implementation of the AggregatedRecord class begins similarly to the Record class, but without implementing a constructor that initializes data members—instead we will initially rely in the zero-values of the member types. The constructor will specify the "way" and the "kind" of aggregation as described in the previous subsection.

For the members, we will choose member names which suit us better for the interpretation of the data. As an example, here is some of the names taken from P2's specification and what we chose to name it:

$$\texttt{numberSYNSrcToDst} \longrightarrow \texttt{srcSYN} \tag{3.2}$$

$$\texttt{numPktsTTL1DstToSrc} \longrightarrow \texttt{dstTTL1} \tag{3.3}$$

Almost all of such adaptations are due to the nature of P2's specification having been around for a while, and having some naming inconsistencies in it. This is relatively inevitable for any software that is used in production but requires new features, and it becomes too hard to change already provided APIs or conventions. For our project, we choose to have relatively less verbose names, and more consistency in our naming conventions.

We will also classify the members in two groups: the 'tags' of a record and the 'measurements'. Tags are those fields which can be used to distinguish one record from another, and will be used to perform `keyBy` operations (described in more detail in the following subsection). They will also be used to check that only records that match said tags can be aggregated: for example, if the aggregation context is origin-to-destination (that is, we are interested in the destination and what are the statistics

of the incoming conversations) and by site, only records with the same destination site tags may be aggregated, and otherwise exceptions will be raised. Flink's distributed engine should prevent any such situations from happening, *provided that the setup laid out by the programmers is consistent*, which is the main reason to implement such guards.

Measurements in the other hand are the data containers for the statistics. They will always be of a numeric type, whether integer (represented by Java's `Long` type) or floating point (`Double`). These are the fields that benefit from the renaming mentioned earlier, and the ones that we will be actually aggregating.

With the fields done, we now must define two operations: what it means to accumulate a Record into an AggregatedRecord, and what it means to accumulate or merge two AggregatedRecords. We will name these methods the same as the methods the Aggregator classes implement, to keep a very close model of the operations.

We will make a distinction between initialized and uninitialized AggregatedRecords, adding a boolean member to keep track of the state. An uninitialized AggregatedRecord can incorporate any other Record, and it will take the all the present tags directly from the incoming record (remember that records may have uninitialized tags, such as a `null` destination site). Otherwise, a check is performed to see if the tags of both records are matched: only the relevant tags will be checked. Taking the same example as before, while aggregating in the context of origin-to-destination, only the destination site tags will be consulted (as any other tags will *not* be guaranteed to be the same). After this check is passed, the data values of the incoming record will be simply added onto the existing values. Here is a summary of the methods:

add (Record) if the current AggregatedRecord is uninitialized, take on the tags of the incoming record; if not, check that all[3] the tags are matching. If the previous step passess, sum all its values to the current ones (the names are translated in this step, like in the example 3.2)

add (AggregatedRecord) if the current AggregatedRecord is uninitialized, take on the tags of the incoming record; if not, check that the relevant tags match, according to the way and the kind of aggregation. If the previous step passess, sum all its values to the current ones.

merge (AggregatedRecord) exactly the same as adding, except the initialized status of both records are assumed.

Time windows

So far we have defined in a rather low level the aggregation mechanism and the classes involved in it. However tailored to the requirements of the project, when it comes down to describing the pieces assembled in the previous section, it is all about summing appropiate fields, and leveraging Flink's very simple aggregation APIs. Here is where we take it one step further, and we will describe what was teased before, the actual aggregation schema.

An aggregation schema is a definition of the core pipeline segment of the application, where the different levels of time window aggregations are specified and laid out in the logical execution graph, in a cascading fashion, that is, providing the correct connections between aggregation operators. We must also define what directions and kinds of aggregation we want to happen at said levels.

[3] All are checked because this is only done in the Raw aggregation, where subnets must match for there to be any aggregation.

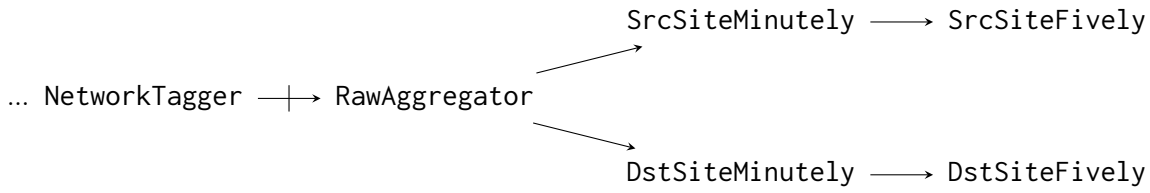

Figure 3.3: Simplified example of an aggregation schema.

In this example, we have the final stage of the preprocessing segment, `NetworkTagger`, and the already described `RawAggregator`. Connected to the ends of that operator we have two lines, and what is pictured are actually `Aggregator` instances, with the operator name set to a differenciating one for ease of later supervision. The `(Src|Dst)-` prefix refers to the direction of the aggregation, and `-Site-` is to distinguish the operator from `-Service-` ones, not pictured yet. Finaly, the suffix indicates the aggregation windows for each operator.

The required Flink declaration-style code for the configuration of the example is listed in the listing 3.1 (the variable '`initialStream`' refers to the output stream of the network tagger). Several things of interest in the code: the first, the use of the `keyBy` directive. As we stated in the introduction to Flink streaming jobs 6, we will have to use different data passing strategies between operators. It is a requirement that aggregation functions have an input that is key-based, that is, the input of the function will be hashed according to a key extracted from the records. In this case, the key extractor can be written as a Java lambda. Notice that in the case of the RawAggregator, the key consists of a string modelling a tuple containing the subnets in the origin and destination and the protocol, and a helper function has been implemented to produce such string.

```
1     // Raw aggregation
2     var rawMinutely = initialStream.keyBy((record) -> record.getKey())
3             .timeWindow(Time.minutes(1L)).aggregate(new RawAggregator());
4
5     // Src - Keyed Windowed Streams
6     var srcMinutely = rawMinutely.filter((record) -> record.srcSite != null)
7             .keyBy((record) -> record.srcSite)
8             .timeWindow(Time.minutes(1L)).aggregate(new Aggregator(Way.SRC, Kind.SITE));
9     var srcFively = srcMinutely.keyBy((record) -> record.srcSite)
10            .timeWindow(Time.minutes(5L)).aggregate(new Aggregator(Way.SRC, Kind.SITE));
11
12    // Dst - Keyed Windowed Streams
13    var dstMinutely = rawMinutely.filter((record) -> record.dstSite != null)
14            .keyBy((record) -> record.dstSite) // Hash by dstSite
15            .timeWindow(Time.minutes(1L)).aggregate(new Aggregator(Way.DST, Kind.SITE));
16    var dstFively = dstMinutely.keyBy((record) -> record.dstSite)
17            .timeWindow(Time.minutes(5L)).aggregate(new Aggregator(Way.DST, Kind.SITE));
```

Code Listing 3.1: Simplified example of an aggregation schema (code).

By setting the key, the aggregating functions know which incoming records to pass through the same aggregation. We fortify this setup by performing some checks at the time of aggregation, in case something was missing, so we can easily find out in a debugging session, rather than having to poke at the internals of the distributed engine and how it distributes the records.

The second method that interests us is the `timeWindow` directive. The function of said directive is self-evident: it indicates to Flink that the following operator must perform it's function in a delimited time window. Flink provides several kinds of windowing mechanisms, including sliding windows and 'session' windows, but for our use case we use the default kind, *tumbling* windows: each window begins inmediately after the previous is finished.

Finally, but not without subtlety, we must notice that the structure of the code has the effect of producing a 'forking' of the data channels, and data is broadcasted along the biffurcation. This happens

at the output of the RawAggregator, when all produced records are sent to all the branches of the following subsegments of the pipeline, and we can detect it both in the dataflow graph as pictured in figure 3.3 and in the code, when we realize that we reuse the '`rawMinutely`' variable to append operators. As we stated in the streaming jobs section 6, this is a dangerous move, especially when we take into consideration that this is only a simplified scheme, and in general we will have four of such lines. However, this duplication is mitigated by the filtering that we apply before each chain, where we want to aggregate only on the records that have actually had a match in the network tagging mechanism (see again 7.2 for more details). Indeed, testing has revealed that this point of the pipeline is not quite the worst bottleneck, so for the time being this is a relatively safe option.

Summary

For a little recap, in this section we have described the way to aggregate and how to lay out the aggregation schema through Flink's DataStream API [KH19, chs. 5,7]. Through the use of custom `AggregateFunctions` that rely on the methods of the unit data classes of Record and AggregatedRecord we provide Flink with the operations required to perform aggregation. By the use of variable representing different streams, the `filter` directive to control the amount of records to pass through a line, the `keyBy` directive to specify the aggregation kinds and the `timeWindow` directive to set the window of aggregation for the following operators we can represent the *cascading aggregation schema*.

Here we depict what the core aggregation schema will look like when we have deployed the full scale application. The default set of time windows will be: 1m, 5m, 1h, 1d. For brevity, we will indicate '`(S|D)`' for 'source' and 'destination', '`(St|Sv)`' for 'site' and 'service' and use standard time units to represent the operator names.

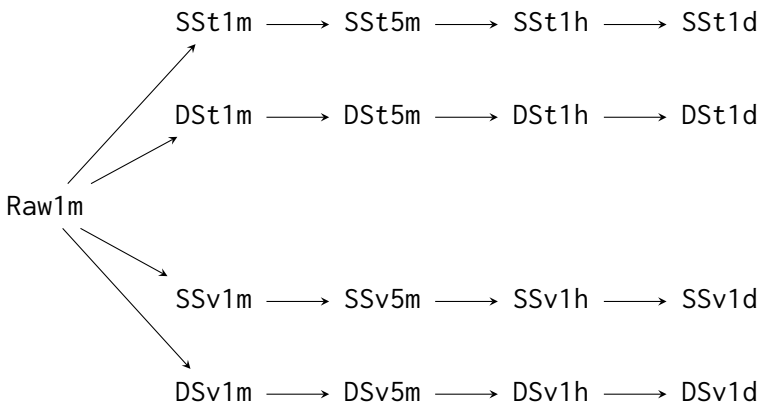

Figure 3.4: Full cascading aggregation schema.

7.4. Rankings

Another of the services that the pipeline must replace, as specified in the formal goals (in section 3), is the *data ranking* process. This is yet another analysis tool that is used to detect anomalies at the site level.

The idea is to use the (by-site) aggregated data streams to produce a ranking of the top sites for each field of a defined set. For example, if we want to produce a ranking based on three fields, like `numConnections`, `srcTTL1` and `dstTTL1`, and we want the ranking to contain the top 100. Then, we need to produce, based on all the records coming from an aggregation window, three lists, each containing the sorted records. In the case of use defined for the development, we will rank the records on 15 fields.

The best match to implement this is the Flink interfaces `ProcessAllWindowFunction`, a lower level API than most of the aggregation interfaces we have used up to this point. When an operator implementing this interface is placed in an execution graph after a time window, it will take all the records coming out of the previous operator in that window, and pass them as a list to its `process` method, which has a deeper access to the execution context than higher level APIs (although we will not be accessing it for this use case). We placed one such function after the Aggregator instances so that they received the record batches directly, and then used the *side output* mechanism to essentially fork the records and send it both to the next level of aggregation and also process them for the rankings.

Figure 3.5: Intercepting the records for the ranking algorithm.

Once all the records are received and the `process` method is invoked, first we output the records so that the next phase of the pipeline receives them. Now, we incorporate the records into a new class to provide separation between the implementation of the Function and the operations done on the record, similarly as we did with the AggregatedRecords. However, in this case, we will use a composition pattern.

RankedRecord and RecordRanker - first approach

The RankedRecord implements a constructor that receives an AggregatedRecord, and then incorporates it as a member, named simply '`record`'. This is done to facilitate operations using the records members directly, as well as to facilitate storage later on (which we will describe in the next section, 7.5). Apart from this, we add a member, `Long` typed, for each field we want to rank against, with the same name. So we have for example '`record.srcTTL1`', containing the actual value from the aggregation, and '`srcTTL1`' with the rank this record holds on the list of records, sorted by the field value.

The first approximation we took was to try to rank the incoming records one by one. To do this, we return to the class implementing the `ProcessAllWindowFunction`. There, we construct a list of RankedRecords, initially empty. In a loop through all the incoming records, we do one of the following:

- if the list is empty, as it will be the first time this loop is run, it adds the record to the ranked list and sets all the 'rank' members to 1
- if it's not, it 'ranks' the record against all the records in the list, updating both the records in the list and the current one. After this second loop finishes, the record is added to the list

The ranking process is done through a method in the RankedRecord class. The `rank` method takes another RankedRecord and compares every field of their embedded AggregatedRecords, and then update the corresponding rank member. To explain the algorithm, let's use a section of the code, seen in listing 3.2.

'`incoming`' is the record passed to the method, so it will be a record not yet on the ranking list, but the one being checked, and '`this`' is the record from the list mentioned previously. Firstly we compare the actual values for the field, in this case `srcTTL1`. Whichever record has the higher value will receive a higher position on the ranking, that is, a lower rank value.

If the incoming record has the higher value, we will demote the current record, as seen in line 7. However, we have to distinguish the case where the incoming record has already passed through this process or not. If it hasn't, that is, its rank member has not been initialized, then we set it to the current's rank before demoting the current record. Otherwise, we will have it either keep its ranking, if it was

```
    if (incoming.record.srcTTL1 >= this.record.srcTTL1) {
        if (incoming.srcTTL1 == null) {
            incoming.srcTTL1 = this.srcTTL1;
        } else {
            incoming.srcTTL1 = Math.min(incoming.srcTTL1, this.srcTTL1);
        }
        this.srcTTL1++;
    } else {
        if (incoming.srcTTL1 == null) {
            incoming.srcTTL1 = this.srcTTL1 + 1;
        } else {
            incoming.srcTTL1 = Math.max(incoming.srcTTL1, this.srcTTL1 + 1);
        }
    }
```

Code Listing 3.2: Code excerpt of the first ranking algorithm.

already lower than the current record's, or give it the current record's. In the other case, where the current record has the higher value, we perform the reverse operations, but we only have to update the incoming record, as the ranking of the current record will not be changed.

This is a rather complicated and hard to follow process. In essence, we are doing a version of bubble sort, and the code listed above is just one part of the sorting that we must understand together with the double loop over all the records and the dynamically growing list. Also problematically, we must implement the listing above *for all fifteen fields* that we want to make rankings with. This is indeed a maintainability problem, but we kept it for a while, as it appeared to be working well.

This, however, proved to be only a failure in our testing environment. We will not go into detail here, as this pertains to chapter 4, but for a brief summary, we were originally testing our pipeline in a test server using a very small amount of real traffic data. Using the same set of P2 files extracted from a production environment, we copied them in a minutely basis, renaming them to make them look newer, and have our source pick them up to process. This turned out to produce a very lower sized stream as aggregations happened repeatedly down the aggregation levels than what a live traffic stream would produce, and thus, each Aggregator produced a number of records low enough that the inefficient system described above did not hurt the performance of the pipeline. As soon as we updated our datasource to have more faithful tests, we noticed almost inmediately a problem that made our entire pipeline and Flink's service crash, and pinned it down to our ranking algorithm. Thus, we gave the implementation another go.

Second approach with Java streams

As we should have expected, bubble sort's efficiency can be vastly improved upon. Instead, we will leverage Java's built-in implementations of quick sort, avaliable for the Java Collections since Java 8. We will also make use of Java Streams, available from the same version.

First, we implement a method on RankedRecord that allows us to update the value of a particular rank to any value, `setRank`, taking a field name as a `String` to indicate which rank we want to update, and the `Long` value we want to set it to. In the method, we unfortunately have to do a lot of code repetition, and check the string in a switch statement:

```
public void setRank(String column, Long rank) {
    switch (column) {
        case "connections":
            connections = rank;
            break;
        case "srcRTT":
            srcRTT = rank;
            break;
        // ...
```

Code Listing 3.3: Switch to check which field to update.

We could have, instead, used Java's built-in reflection capabilities, which include reflective methods to obtain the class members by member name, which would simplify the code significantly. However, the performance of such an approach could have been very poor: as stated in Java's Documentation on the reflection API [Ora19]:

> *Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.*

This is very much the case for this section of the project: this method will be called too many times to leverage this simplification, as it would render other optimizations futile.

Now, back to the RecordRanker. This time, we will produce fifteen copies, making sure not to do a shallow copy (for example, by simply pointing another pointer to the original list), of the received records, one per field to generate a ranking from, as described at the beginning of the section. For the time being, it is fine that we work with AggregatedRecords instead of RankedRecords.

A very concise way to express the production of a copy of the list for each ranking field is to use the Stream API, in particular, the `map` function: we can produce an array of all the field names we want to make rankings with, and then turn it into a stream. Calling `map` on that stream, we can pass a function that takes a field and produces a copy of the list of records as described just above, and then sorts it based on the field name.

```
switch (field) {
    case "connections":
        copy.sort((AggregatedRecord r1, AggregatedRecord r2) ->
            -r1.connections.compareTo(r2.connections));
        break;
    case "srcRTT":
        copy.sort((AggregatedRecord r1, AggregatedRecord r2) ->
            -r1.srcRTT.compareTo(r2.srcRTT));
        break;
        // ...
```

Code Listing 3.4: Custom comparator functions for `sort`.

We make use of the methods available to the numeric types `Long` and `Double` to compare the values. The negative sign in front of the comparisons is because the default behavior of sort, which uses comparison functions such as this one, is to order in ascending order, but we want to have the higher valued records first, so we invert the comparison result. This could be adapted if there was a field that had to be reverse-ordered, or if there was a special ordering to it, but so far that occasion has not arrived.

At the end of the `map` function we return a `Tuple2` containing the field name with which we sorted and the sorted list, trimmed to 100 entries, giving us the actual tops of AggregatedRecords. This is a

simple truncation, in that we don't take into consideration possible duplicates, but for the general requirements it is acceptable. Now, we continue to write in the functional stream style, and use a `forEach` function to operate on the tuples. For a start, we transform each sorted list of AggregatedRecords into RankedRecords, setting their rank accordingly (this is why we pass the field name, so that we may call `setRank`) with their position on the list.

What we want to do now is merge the fifteen lists into one list, merging records that appear in mutliple rankings: say a record of the batch appears in three of the rankings. Then, in the result list, we will want to have only one RankedRecord with the same underlying AggregatedRecord, and all three rank members set. We do this by building the merging list including all records of the incoming lists as they come into the stream, and then checking to see if the record keys match, and if they do, we update the record that was already added with the rank of the matched record.

`srcSite`	`dstSite`	`numConnections`	`numConnections` (rank)	`srcTTL1`	`srcTTL1` (rank)
A	A'	128	0	0	-
B	B'	90	1	49	7
C	C'	12	-	199	0

Table 3.1: Example list of merged RankedRecords. Notice not all of them have rankings set, but all of them have at least one set, hence they belong in at least one ranking.

We could do this in the storage layer performing an "upsert" operation where we either insert records or update their fields if we detect that they are already inserted, assuming compatibility with this feature in the storage layer appended after the ranking. This would be indeed a simplificaton, getting rid of the need to merge the lists, but it is a fair tradeoff: given our constraints, the lists of RankedRecords will (generally) always be 100 elements long, and without merging this will mean that we always upsert 1500 lines to the database. By merging, this could reduce to a minimum of 100 records, which is highly unlikely (as this would only happen if there are 100 records, that is, sites, that have the top 100 position in all the metrics ranked), but this way we limit the size of the interaction with the database, which is always a nice optimization.

Summary

With a direction agnostic record ranking function, we decorate each node in the by-site chains of aggregation of the pipeline (top two lines in figure 3.4) by attaching the class as the inmediate operator and using both the normal and side outputs to effectively pass the original data and also generate a ranking for the aggregation window (as seen in figure 3.5). So far, this brings the number of operators in the aggregation segment of the pipeline up to 25 (1 raw aggregator, 16 aggregators and 8 rankers). We will give a *small* sketch of what the execution graph looks like in the end, although even now it would be quite difficult to display it in the document.

7.5. Persistance

So far we have laid out the aggregation logic of the pipeline, and the tests with very simple sink functions (such as a simple printing to standard output sink) provided by default in Flink show consistent results. However, we need to get all the information that has been produced at every aggregation level and keep it so that we may later perform analysis or simply visualize it. For this task, we will implement "materialization" classes for the two kinds of records that get to the end of the pipeline that will act as the datastream sinks. For the implementation at hand, we develop PostgreSQL adapters (or 'ports', if we think back on the Hexagonal Architecture terms [Coc05]), although some more sophisticated ideas have been considered, such as writing text reports of some format, which would be a relatively trivial matter programming-wise (not so in a document design sense). But before getting to that, we introduce the mechanism supported by Flink that we will use for the enforcement of exactly-once guarantees.

Write ahead logs (WAL)

There are two fundamental mechanisms integrated in Flink that would allow a sink to provide exactly-once guarantees: a two phase commit (2PC) system, or our choice, a write ahead log (WAL). For more detailed descriptions of their differences, one can look at the documentation [Fou19a] or the reference text [KH19, ch. 8].

A WAL system integrates with Flink's checkpointing mechanism, briefly mentioned earlier in this chapter. From the reference text:

> The WAL sink writes all result records into application state and emits them to the sink system once it receives the notification that a checkpoint was completed. Since the sink buffers records in the state backend, the WAL sink can be used with any kind of sink system.

While WAL sinks do not give the exact guarantees as the 2PC system would do, it is a relatively simple system to implement, and together with a reliable system to write against, such as Postgres, it gives enough reliability. To implement it, we need to extend the `GenericWriteAhead` class, parametrized with our record of choice, and implement an extra class, a `CheckpointCommitter`. This class will be used behind the scenes by Flink to coordinate with Postgres as per to when to commit transactions, using an auxiliary database schema.

PostgreSQL

The most important part of the implementation of a materialization class, though, is the required override of the `sendValues` method. To this method, a list of the incoming values is passed, together with a checkpoint ID and timestamp which we will not use (as we already have the timestamps on the records as members, and the checkpoint would only be useful if we were using some other writing system other than Postgres).

For our implementation of this class, we use Java's SQL standard package together with the JDBC driver to connect to Postgres. Upon the initialization of the class, at the beginning of the pipeline lifecycle, there is an '`CREATE TABLE IF NOT EXISTS`' statement that verifies that there is a table in the database corresponding to the current level and kind of aggregation. The table names are passed as a constructor argument, and we will provide them systematically upon deploying. Afterwards, only the `sendValues` method will be invoked.

First, we establish a connection to the database and turn of auto-commiting of statements. Then what we do is create a prepared statement, a standard feature of SQL, where we can reuse an SQL statement, modeled in the code as a variable of `PreparedStatement` type, and then we can loop over the provided list of records and for each one we can set the placeholders to the values stored in the records, shown in listing 3.5.

```
1  PreparedStatement stmt = connection.prepareStatement( "INSERT INTO "+table+" VALUES ("
2      + "?," // timestamp TIMESTAMPTZ
3      // ...
4
5  for (AggregatedRecord record : records) {
6      stmt.setTimestamp(1, new Timestamp(record.timestamp)); // timestamp TIMESTAMPTZ
7      // ...
8  }
```

Code Listing 3.5: Usage of an SQL prepared statement with incoming data.

After setting the values, the prepared statement is added to a batch of statements, another feature provided by Java's SQL package, and we continue until all records are processed. After this, the batch is executed and we tell the connection to commit the changes.

We implemented an `AggregateMaterializer` and a `RankMaterializer`; these classes share some of the structure of the statements as in the example code, but the tables where we will write rankings

will have, in addition of all the columns of the other tables, a column named '<field>_rank' for each regular column, where we will store the integer values of the rankings. There is also a RawAggregateMaterializer, corresponding to the obvious node, which has the specificity that it must include all the tags that the other tables can ignore, as it records site-to-site conversations, where the rest simply aggregate in a direction per-site or per-service.

Some amount of error handling is done here, mostly around connectivity issues, but since the database is outside the range of action of this project, further guarantees must be provided by the operators of the software and the system administrators of the servers it will run on. This includes auto-deleting mechanisms for old data: even though we reduce the data rather dramatically in size after the aggregations, especially in the later stages of the pipeline, there is still a very large amount of incoming records, and every minute more. Disk space is very much a critical issue with this project.

As for the deployment of these sinks, we will attach one to each regular aggregator and ranking operator, as well as the raw aggregator, each of the corresponding class. The table naming schema is as follows:

```
p2_raw
p2_(src|dst)(site|service)_(minutely|fively|hourly|daily)(_rank)?
```

and as expected, this will double the number of operators in the fully functioning aggregating schema described in 3.4 and expanded with the ranking operators, which had 25 operators already. This also induces a broadcast data passing strategy between some of the nodes of the graph that are not leaf nodes:

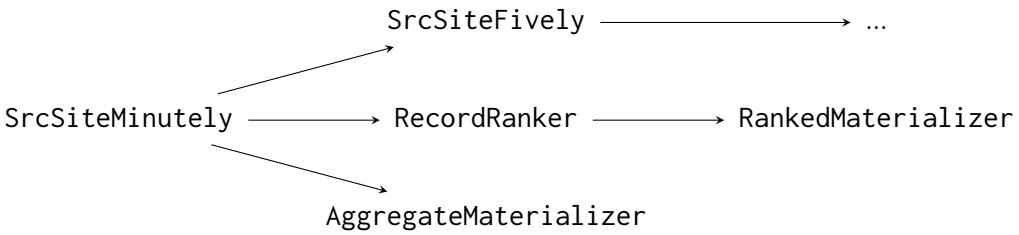

Figure 3.6: Zoomed in section of the full powered pipeline.

As we can see, the sections closer to the beginning of this segment can get quite heavy in computation and stream sizes, as they have a large influx of minutely data. We will make some remarks on this in chapter 4.

This concludes our description of the Flink side of things. Save for some intermediate classes that perform small jobs here and there, this section contains an in detail explanation of the design of the Java classes that Flink runs in this project, as well as the way in which we integrate them with the DataStream API. For a relatively interesting picture of the job graph, see figure B.1 in the appendix: the job is too big to show a decent presentation of it in detail, but the general aspects we described can be appreciated.

8. Grafana

Now that we have implemented a replacement processing pipeline, complete with database integration, all that is left to complete the project is to extend our replacement efforts to the Grafana visualization layer, with the same principles in mind: end product equivalence, maintainability and efficiency.

As we recall from chapter 2, section 4.5, we will be focusing on two dashboards, of which one will be "duplicated" in a sense: the first is for ranking data, or 'tops' as we will refer to them, and the other two are for aggregated data, one for by-site aggregation and the other for by-service. These dashboards are already structured in the current implementations, so we will take their layout for granted, and focus only on improving the data querying mechanism.

Queries

Grafana provides official support for a number of datasources, among which is Postgres. Once a running instance of a Postgres database is registered in Grafana, panels in dashboards can select it as its datasource and provide an SQL query. All the panels we will use will attempt to plot time series data, that is, it will plot a numeric value as a function of time, possibly with a tag per datapoint (see the appendix for an example in B.2).

With a Postgres database, this data has to be provided in a particular format, that is, the result must fit into a table with three columns:

1. one with the timestamp, to place the data point in the horizontal axis
2. one with the numeric value, to give it its vertical axis position
3. one with a string value, which will give a datapoint its tag

These columns need not be provided in this order, as long as the three have distinct types matching what Grafana expects, or we can provide SQL aliases to the columns in the result for extra clarity: respectively, "time", "value" and "metric".

To facilitate some of this interfacing, Grafana has a mechanism of variable expansion and macros. In a query, a special token can be inserted, either starting with `$` or surrounded by `${}` or `[[]]`. The rest of the token is a variable name, which Grafana allows us to define in a dashboard, or a macro. For example, we could define a variable named `$limit`, taking values 5, 10, 15, 20, 50 or 100, and use the macros for Postgres `$__time()` and `$__timeFilter()` to produce the following query:

```
1  SELECT
2      $__time("timestamp"),
3      srcttl1,
4      site
5  FROM p2_srcsite_minutely_rank
6  WHERE
7      $__timeFilter("timestamp")
8      AND srcttl1_rank IS NOT NULL
9      AND srcttl1_rank < $limit
```

Code Listing 3.6: First example of a Grafana adapted SQL query.

This query will give us a table with a timestamp column, which by the macro in line 2 is taken from the column named 'timestamp' and aliased as 'time', and Grafana uses the types of the other columns, `DOUBLE PRECISION` and `TEXT` respectively to detect which column is the 'value' and which the 'metric'. The `WHERE` clause does three things: it limits the query to only include records with a timestamp included only in the currently selected time range of the visualization, it drops records that are in the ranking table but not for being in the top 100 by their `srcTTL1` field value, and lastly it limits the number of records returned to be dynamic, and the user of the panel, when choosing one of the values for the variable, will see only the number selected. Notice that this was the reason to include an embedded AggregatedRecord in the RankedRecord: that way we can look only in one table to consult the actual value of the desired fields while having only a small amount of records to process, all guaranteed to be in the ranking.

This is a fine first approximation, but there are a lot of improvements to be made. Although Grafana is well adapted to this situation, it also requires that the data is sorted by time and value, as well as

grouped together. These are simple fixes to the above query, but then we encounter some issues like Grafana having trouble to process what are essentially multiple time series, one per site, that start or end in the time range, because some sites appeared only for a little time in the ranking. That fix is not trivial, but it does not concern too much the matter at hand, so suffice to say that our approach to solving this problems is to hide the complex queries behind PL/pgSQL functions (sometimes called stored-procedures: the same thing), which then can be easily used in panel queries (code listing 3.8).

This also makes it easier for technical users or support of the dashboard to change queries without having to have a deep knowledge of SQL.

With this approach, it is easy to see that the two different kinds of dashboards will require very similar functions to operate. For that reason, there are two functions produced, one as seen just above, `p2_ranked_series` and two more named `p2_site_series` and `p2_service_series`. They will share a lot of the code, and in fact that is factored through the use of more internal functions not intended for the end user, but they will select from the correct columns in each case. The ranked series function will have the limiting characteristic we have seen so far, but the aggregated series ones will behave a little differently. The idea is that selecting from all records in an aggregation window is too much, but a user can pick which sites or services they want to display, with a certain limit of number of choices, and that way compare their trends simultaneously. This is done using a very useful SQL feature, the `ANY` operator, in conjunction with the multiple-selection variable mechanism from Grafana.

A variable taking values in the different sites or services (which are provided through an external source, in our case the CSV file mentioned in the NetworkTagger section 7.2) named `$sites` is defined, and then the following variable interpolation token is passed to the function: `'{${sites:raw}}'`, which, with three sites selected, for example, interpolates to `'{{SITE-A,SITE-B,SITE-C}}'`. Then, in the `WHERE` clause, we can add a line like: '`... AND site LIKE ANY(%L)`', where we will pass the interpolated value as a literal string, and this will equate to finding all records where their site tag is one of the three values. These results will behave in very much the same way as the ranked series did.

Dynamic table select

One thing that pertains very much our efforts to provide an analysis-first approach to the visualization layer is what we do with the density of data shown. Imagine trying to obtain data of the last week trends in traffic, consulting the ranking data, but we use the same table as in example 3.6. The user would then have to wait for *all the records stored to be recalled, and painted*, and that means loading the records for 600k+ minutely slots, each with possibly a huge amount of records. This is, of course, not acceptable, and in fact that is the reason why the latter aggregation stages exists: to condense data trends so that the huge volume of information is processed before visualization.

```
-- The function arguments or variable will be named _name_
-- ...
begin
    _table_ := case
        when (_direction_ is null or _start_ is null or _end_ is null) then
            null
        when _range_ < interval '1 hours' + interval '5 minutes' then
            coalesce('p2_' || _direction_ || _agg_kind_ || '_minutely')
        when _range_ < interval '6 hours' + interval '5 minutes' then
            coalesce('p2_' || _direction_ || _agg_kind_ || '_fively')
        when _range_ < interval '3 days'  + interval '5 minutes' then
            coalesce('p2_' || _direction_ || _agg_kind_ || '_hourly')
        else
            coalesce('p2_' || _direction_ || _agg_kind_ ||'_daily')
    end;
    return _table_;
end;
-- ...
```

Code Listing 3.7: Dynamic table select function.

Our solution to this problem is to change dynamically what table data will be selected from. To do this, we write an internal function, used in the wrapping functions for queries, that returns the table name as text (code listing 3.7)

While it looks complicated, the function is just a `CASE` clause, equivalent to a switch in other programming languages: it takes the time range and tries to fit it into an interval. Notice the 'jagged' pattern of the `WHERE` clauses and which table is returned, and recall that we had a very similar pattern happening in table 2.1. This is done to fit a relatively similar density of data regardless of the time range selected, or, since that is very hard (or impossible), at least mitigate the effect of the selection on the amount of data.

Figure 3.7: On the top, a time range of an hour, and on the bottom 24 hours.

In figure 3.7 we can see the data density disparity, but it is not terrible: in the daily information dashboard, the last 24 datapoints from the hourly table are selected, and in the hourly case it is selected from the minutely table.

This table selection is integrated into the different functions to retrive time series, as exemplified in listing 3.8, so that the editor of the dashboard does not need to be concerned with selecting the right table.

```
select
  time, metric, value
from p2_ranked_series(
  'srcttl1',            -- The values we want to see.
  'src',                -- The direction we want it in, meaning from p2_srcxxx...
  $__timeFrom(),        -- The start of the current time range
  $__timeTo(),          -- and the end
  $limit                -- The number of records to retrieve.
)
```

Code Listing 3.8: Example of the usage of a stored procedure for querying.

Because of the lack of space in the main body of the document, images of both of the dashboards are included not here but in the appendices, see figures B.3 and B.4. Site names have been blurred to protect the client's sensitive data, but the main features of the aggregation are shown.

4
Tests and Results

In this chapter we will look into some of the results obtained by doing this tech stack replacement. Most of the measurements will be obtained with the Prometheus monitoring tool, for which Flink has official support, but certain comparisons will be hard to make against the previous software. In those cases, approximate numbers will be given when appropiate, but we will focus our attention on different aspects of the comparison, along the lines of our stated goals.

9. Testing environments and subsequent results

Initial testing at low flowrates

Initially, we used a testing environment consisting of real, but limited, P2 output files that were repeatedly copied by a script into the directory that the InotifySourceFunction was pointed to, changing the timestamps to pretend that it was the real P2 output. For the tests, we were copying a random choice from a set of files with file sizes that were fairly representative of the average, around 120Mb, and somewhere between 100K and 160K lines per file, giving us an estimation for the records per minute. As mentioned in the rankings section in the previous chapter (7.4), this was mostly a good test for throughput of the preprocessing and raw aggregation stages of the process, since everything down the line would get dramatically reduced in size due to the repetition of files.

Indeed, from the early tests it was clear that the fully deployed pipeline with a parallelism of 4 task operators by default (the exceptions being the initial source of Inofity events and the materialization sinks) was able to process all the records in time before the next batch arrived. This means that the heaviest operators, the RawAggregator and the NetworkTagger, were performing fine. From the rest of the aggregation schema, only the minutely and perhaps the fively aggregators were of concern, as the rest of aggregators were receiving records in a very reduced rate, and furthermore they had ample time to process a batch before the time window closed. Even so, these operators also worked with a reduced set of records and performed just fine. The record rankers performed well as well, even though at this stage we were using the first iteration of the ranking algorithm (see section 7.4).

Closer to real-time data tests

After we switched the testing setup to a more realistic environment, things changed slightly, however. This new setup consisted in importing a number of network traffic traces, captured over the span of a day and a half in a real production environment, with a total size of just over 800G. This traces were then replayed as if they were real traffic, and the P2 tool performed its function over this stream of packets, and finally we pointed the source of the pipeline to P2's output, giving about the same average file size, but this time producing not a single file but actually three files per minute, the three protocols we ended up implementing (TCP, UDP and OTHER, in P2's classification). This not only increased the initial throughput, but it also made the reductions obtained by the aggregations smaller, as they were

not only processing more data, but the data was more heterogeneous, making the chances of any two records being in the same aggregation lower than in the previous setup.

The first thing that we had to do was turn off the latency meters that we had been testing, as they were introducing too much stress on the operators, meaning that we were not getting very reliable metrics from it. Even so, we still had many metrics to analyse the performance of the job, so that was not a big problem.

But then we noticed that after a while, every instance of the job seemed to be failing and not be able to recover, despite Flink's best efforts to restore the job. As we investigated, it became clear that the source of the problem were the ranking operators. Indeed, when we looked more carefully, the slowness of the original algorithm made it take too long to output anything, so it was caught up with the minutely checkpoint operation. This checkpointing had to take all of the state of the operators and back it up, and that started taking exponentially more time as the ranking algorithm could not catch up and that resulted in the checkpoint having to back it up plus the accumulating records in its queue. This meant that eventually the checkpoints failed to finish, and in that state Flink was unable to restore the pipeline.

Then we implemented the second iteration of the ranking algorithm, and indeed everything went back to operating as smoothly as before. In general, records were being ranked in an average of 15-20ms accross the different levels of aggregation, clearly indicating that the rankings would not become a problem anywhere. The initial segments of the pipeline had already passed the test of performance, and having fixed the problematic ranking algorithm, everything else passed the test as well.

Data structures vs. class members

At some point in the development process we tried to model our Record classes using a Java `Map`, because we did appreciate the code repetition problem that arose whenever we had to do the same operation over all the fields of a record, and wanted to condense those actions in a smaller part of the code that could be iterated. However, as much as that would be a huge improvement to the maintainability of the code, there were some problems to consider:

1. The performance of having an intermediate representation of the data may not seem much of a problem, especially if the key space of the field names is small. However, one has to take into account the rather huge amount of records that the pipeline will need to process every minute, and any small price to pay in a single field operation must be multiplied by that factor. In a critical project such as this, where other solutions have proven to fail under the work load, having the compiler optimizations of class members provides an assurance too good to give away.

2. The data that is to be stored in these fields does not have an homogeneous type: rather, it will be of integer or floating point type. This does not seem to be much of a problem either, at first, but Java does not have union types which would allow the values stored in the Map to be of different types. Other solutions, such as having tuples as values, would require that we have a 'guide' to indicate of which type a field is, and since we *want* that to be dynamically decided, as that is the whole point of this mental exercise, that would require an additional lookup of such guide... defeating the whole purpose by the same reason of performance as before.

3. Flink requires that all the members of an object passed through its operators be storable in its state backend, and there is a whole section and much to be said about how Flink goes about serializing and saving its state, as in chapter five of the reference text [KH19]. Since we chose Java as our language for the application, our choices are limited to primitives (the standard types of Java, such as `Integer`, `String`, `Double`...) or POJOs, that is, Java objects. These objects must be composed of only primitive types, or provide serialization information so that the checkpointing is able to optimize it's storage. By using a Map instead of the currently used primitive types, we would sacrifice some of the performance of the checkpoint process (which, as we saw with the first ranking implementation fiasco mentioned before, could turn out to be critical) or we would have to provide a clever way to optimize it, which is not a clear cut task.

10. Comparisons

Performance

In this section we will not give exact numeric comparisons as we are unable, not because we could not monitor our solution's performance, but because we could not monitor very well the previous solution. As these were more complicated setups, and none of them were properly instrumented in the first case, we had to go by ear in order to provide a relatively reasonable comparison.

As is done in our replacement job, the original AWK scripts are run in two fashions, the "raw" aggregation mode and the rest of regular aggregations. With the same inflow of data as in our testing, the main raw aggregator takes about 20s to process everything using a single core at max capacity and write the records to the corresponding raw table, but beyond that we can't give too much information on what amount of CPU or memory usage of the rest of the scripts. This is even harder to compare to our pipeline as the parallel task operators are reflected as threads, and the total number of operators goes up to 180+ in the fully deployed pipeline, and Flink manages them transparently to us.

Thus, in this regard it will suffice to say that Flink passes the performance test in a close-to-real environment, running in the same system as all of the other Naudit tools, and Grafana, Postgres, etc. In the appendices there is an extract of the Prometheus metrics of CPU load and memory usage by the tasks, reported by the deployed Flink's task manager (B.5). As this is the same witht he previous stack, that is, we can only tell if it's performing well enough or not by evidence of results in the database or their absence, we will call this comparison a success, as no problems regarding performance where observed after fixing the most critical bugs.

Monitoring

In this regard, as is evident, the Blink development has greatly outdone its predecessor. Not only by the available metrics internally, as we have sprinkled in the document, but by the level of monitoring granularity that Flink allows. Whereas the previous stack was fairly impossible to monitor in a per-unit of computation basis, as instumenting everything is a dauntingly big task in size and also compatibility, Flink allows us to monitor many things in a per-operator basis: average latency, incoming records 'backpressure' (meaning the amount of time records spend waiting in an operator's queue before being processed), throughput in bits *and* records per second, as well as all the Prometheus metrics exposed, but regarding each operator. We show some examples of this operator-level monitoring in the appendices—see figure B.6.

Not only that, but Flink's logging mechanisms gather all of the operators logs and provides them in an organized fashion, which means that we can identify failures and the *original causes*. By following the logs, we were able to quickly identify and fix the problems as the development went ahead.

Operation

This is another great success of the project. Flink's operation is as simple as a call to a binary CLI interface, pass it the streaming job compiled JAR file and the required parameters, and it launches everything transparently, registering it to the monitoring website and updating it's metrics providers. Through the previously mentioned monitoring, provided by default, it is made really easy to set up alerts or even check out at any given time the status of the job and the system, giving many more opportunities to detect failure on time and resolve issues.

On the other hand, while the execution of the previous solution was also condensed into one coordinating script which launched and set up everything else needed, from there it is a loose system, and not at all easy to monitor. The effort it would take to instrument the code by adding connectors to the usual monitoring tools would be complicated, and even if it was not, it would still need doing, as opposed to our system which already has done this job.

Maintainability

And finally, one of the most important goals of our project was the maintainability of the software. This will not be directly compared, as the code for the previous system is owned by Naudit and we cannot make comparisons in good faith. However, as has been proven by the actual development of the project, the logic is easy to read both at the high level of the layout of the jobs, and down at the level of the aggregation, ranking and storage levels. We were able to implement simple changes to all parts of the project with relative ease, such as when we solved the rankings problem, but also when we modified some parts of the Records internal logic, which had the requirements changed several times.

The places where code is more complicated are well commented or are well tested, and new changes and implementations are easy to write as we have working examples, and also we have Flink's system as a guidance on how to model what we want to achieve. Not only that, but as we discussed, even if parts of the fundamentals of the data changes, the middle layer of aggregation classes will be able to be reused without even having to touch them, or the general job arrangement. This is a very good example of modularity and separation of concerns and their effect in well produced software.

5
Conclussions and future work

11. Conclussions

As a principal conclussion of the work done for this project, we can say that the adoption of modern and powerful systems such as Apache Flink are a fantastic way to organize and improve a project where the core logic is simple but the orchestration has gone out of hand. By sticking to the guidelines of a sensible programming model that provides most of the background work of organizing executions we can concentrate in getting things right at the core level, without sacrificing too much performance to the more "closer to the metal" solutions, and in some places even gaining some advantage.

Simplification of the most important parts of the applications is key. When the upstream information that a system consumes has a complicated, out of date or simply missing some information, adapting it to a more natural model that is easier to read and reason with is just as important as processing it right: the goal of a maintainable project is that future developers who may join in the effort of keeping the software up to date must have an easy time understanding the model so that they may get their hands on it quickly, but most importantly, correctly. Our project implemented simplifications for most of the processes involved in the pipeline, but even then it could benefit from more work in that regard, such as the unification of data representation classes like our Records.

Not all is positive: proper support for data structures is of the highest concern when developing a streaming (or any) application, and in our case, Java falls short. This is evidenced by the amount of repetition we had to implement in many parts of the code, because we had to operate on all relevant members of an object. In other languages, such as JavaScript, we could simply iterate over the keys, and in laguages with better data structure support we could do the same. Instead, as it is we must provide with an alternative implementation if we want to avoid code repetition when further developing our project.

12. Future work

12.1. Robustness

As mentioned in the 'Data ingestion' section 7.1, there are some flaws regarding possible loss of data with our implementation of the `InotifySourceFunction`. As it stands, it is possible that the pipeline fails without notice, and there is no mechanism to detect whether there are input files that have not been processed. This should be addressed to provide a more robust support in a production environment. For example, we should take note of all the files present in the watched directory and checking against a list of already sent filenames, and send the corresponding missing files in order. This would require a mechanism to indicate to Flink that this 'late events' belong at a different time, as well as a thorough testing period.

In the same regard, more effort needs to be made in order to guarantee that the jobs will be able to be restarted from savepoints and recover from puntual failures. While this has been the case for some small amount of detected failures, such as with the Postgres integration (where Postgres refused some connections due to reaching the limit of clients at some times), and the related operators have proven to recover without anything important to notice, the more serious crashes have been impossible to resolve gracefully, forcing a restart of the system and the job (with the inevitable data loss).

12.2. High availability

Some parts of the project may seem incomplete, such as the lack of data ranking in the 'bidirectional' sense. This too is for a good reason: while for now Blink performs well in good conditions, the addition of a huge chunk of processing nodes to it could prove to reach the limit of the server capabilities. This could be solved by attempting to run the pipeline in separate machines, clustered under the Flink distributed system, for which heavy testing is necessary, as well as a correct setup and possibly some changes in the code.

12.3. Generalization: recipes

We described in the conclussions some of the negative take-aways of the project, such as the excessive code repetition. An idea for an automated code generation solution was floated during the early stages of the project, but it was left out due to size and time constraints. Here we present what we would have desired from such a system, and how to address the problem.

The first important note is that this solution should be intended not for experienced developers only, but to be accessible to a broader audience, such as analysts and junior developers. We would like to have a simple 'recipe' specification where one could express the way the data should be laid out in the Record classes, and also which aggregation schema is desired, possibly changing the amount of levels, their time window durations, the different tags on which to aggregate...

Obviously, such a system can not be omnipotent, or else it would be easier to just write the program for Flink directly. For that reason, we would limit the users choices to the following restraints:

- Only known datasources may be used. In this case, only P2 is available, but once a Netflow or other datasource is implemented, it could be used as an intake.
- The changes to be made to the aggregation schema can only be related to the amount of lines desired, whether the lines must produce rankings or not, and the levels of aggregation. The same cascading schema would be kept for the logical increasng order of the levels.
- The aggregation processes can only be simple operations, such as sums, multiplications, etc, from a set of provided operations, and the user may specify which fields of the datasource they are interested in aggregating.

A similar approach could be taken to produce the necessary queries and functions in the SQL side of things, so that all the necessary code could be integrated into dashboards and be exactly in accordance to the aggregations without having to use a developer's time to set up the dashboards manually.

For this to work, we would need to be able to produce source code that can be used by the Flink system. There are a number of templating engines that could be used, among which we considered Go's template language[1] and FreeMarker. We would have to use the code examples already produced and then extract whatever logic we decide to give to the recipe system, and make it so that there are excerpts of it that can be populated only with the user data. We would need to program a consistency checker for the data, and surely some logic to be able to write simpler templates and organize them in a way that they can be directly passed to Flink as a compilated streaming job.

[1] https://golang.org/pkg/text/template/

A
Source Code

```
DataStream<Tuple2<String,Integer>>dataStream=env
// The datasource, in this case a socket, listening on port 9999
.socketTextStream("localhost",9999)
// Converting incoming text blocks into tuples of non-empty words and the
// integer 1
.flatMap(new Splitter())
// Hashing the stream taking as key the first element of the tuple
.keyBy(0)
// And indicating that this operation is to be performed in 5 second windows
.timeWindow(Time.seconds(5))
// ... and after the window closes, aggregate with values taken from the
// second element of the tuples
.sum(1);

...

public static class Splitter implements FlatMapFunction<
    String, Tuple2<String, Integer>
> {
    @Override
    public void flatMap(String sentence, Collector<Tuple2<String, Integer>> out)
    throws Exception
    {
        for (String word: sentence.split(" ")) {
            out.collect(new Tuple2<String, Integer>(word, 1));
        }
    }
}
```

Code Listing A.1: Flink streaming job example from the official website.

```
public AggregatedRecord add(AggregatedRecord value, AggregatedRecord accumulator) {
    accumulator.add(value);
    return accumulator;
}
public AggregatedRecord merge(AggregatedRecord a, AggregatedRecord b) {
    return a.merge(b);
}
public AggregatedRecord getResult(AggregatedRecord accumulator) {
    return accumulator;
}
```

Code Listing A.2: Simplified Aggregator logic.

(This code is without exception handling, which is required as the methods implemented in the AggregatedRecord class may throw errors, but it is presented as such for presentation's sake.)

B

Graphics

Figure B.1: Fullest view of the full streaming job provided by Flink's job monitoring web app.

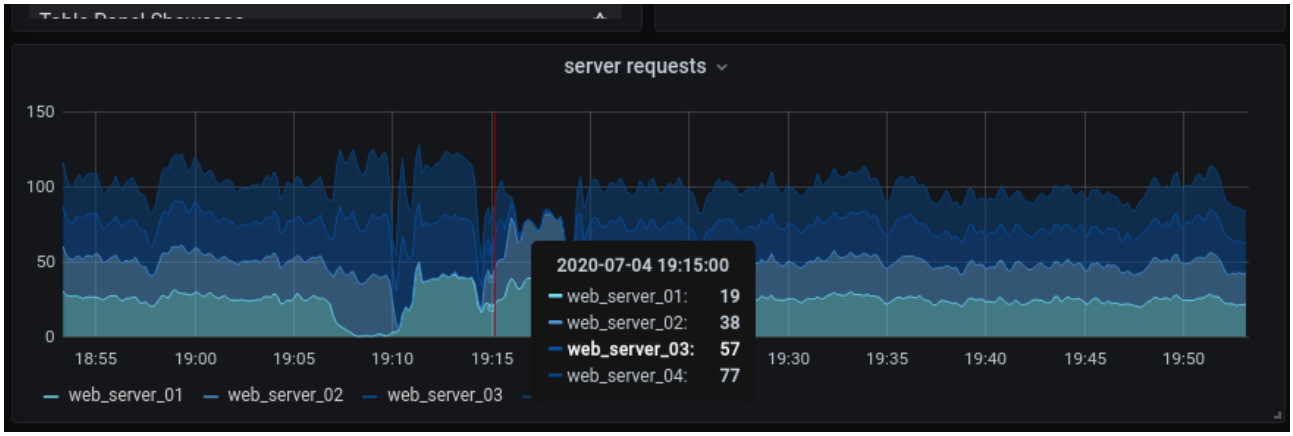

Figure B.2: Example of a Grafana “Graph” panel, with tagged datapoints.

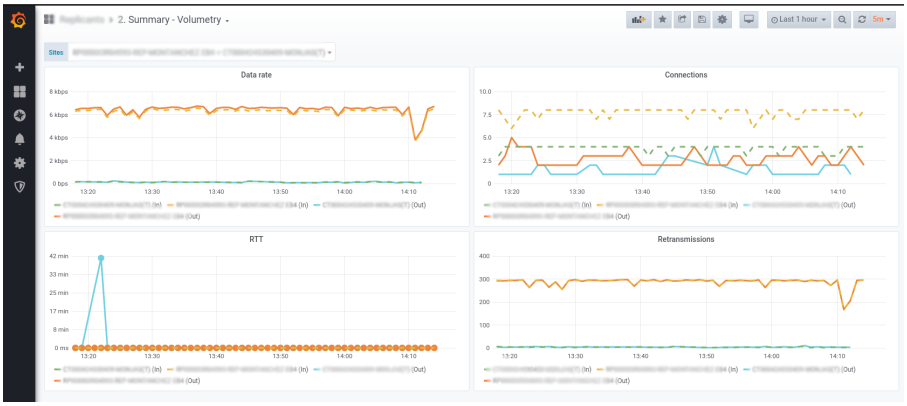

Figure B.3: A small version of the per-site selection of aggregated metrics, with two selected sites.

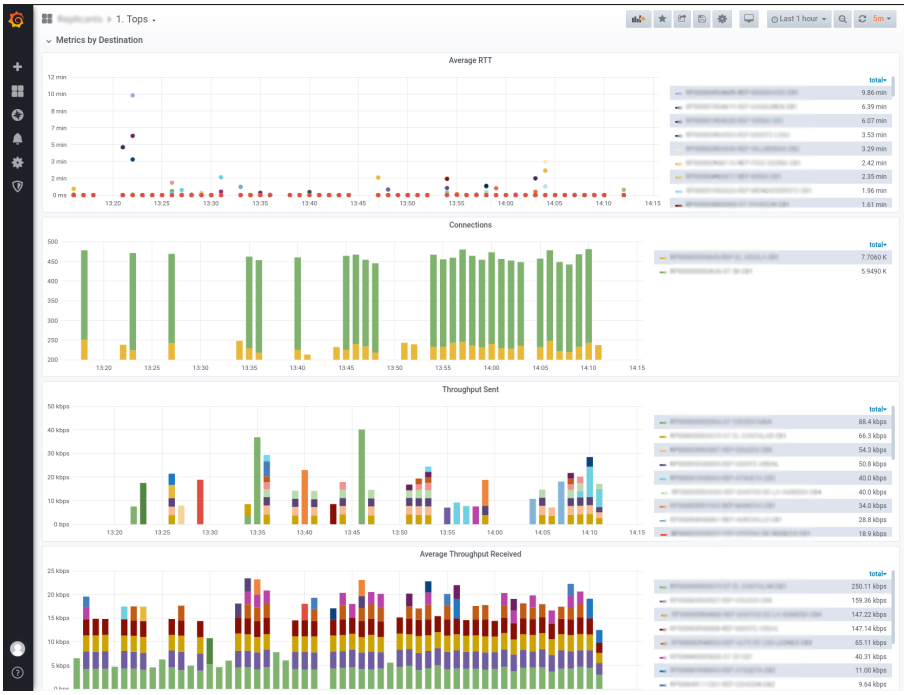

Figure B.4: A section of the ranking data dashboards.

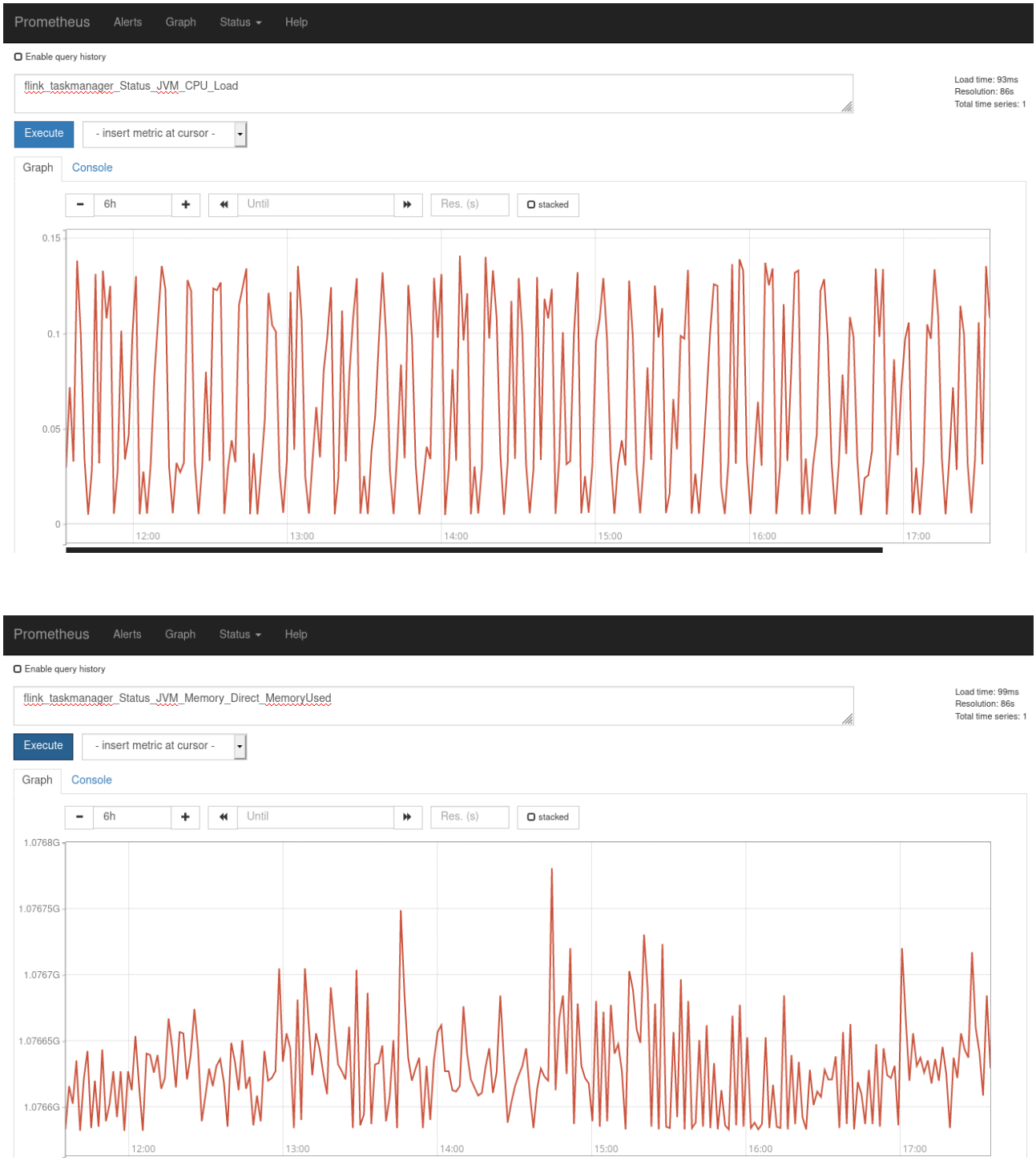

Figure B.5: 6h extract of Blink's CPU load and memory usage

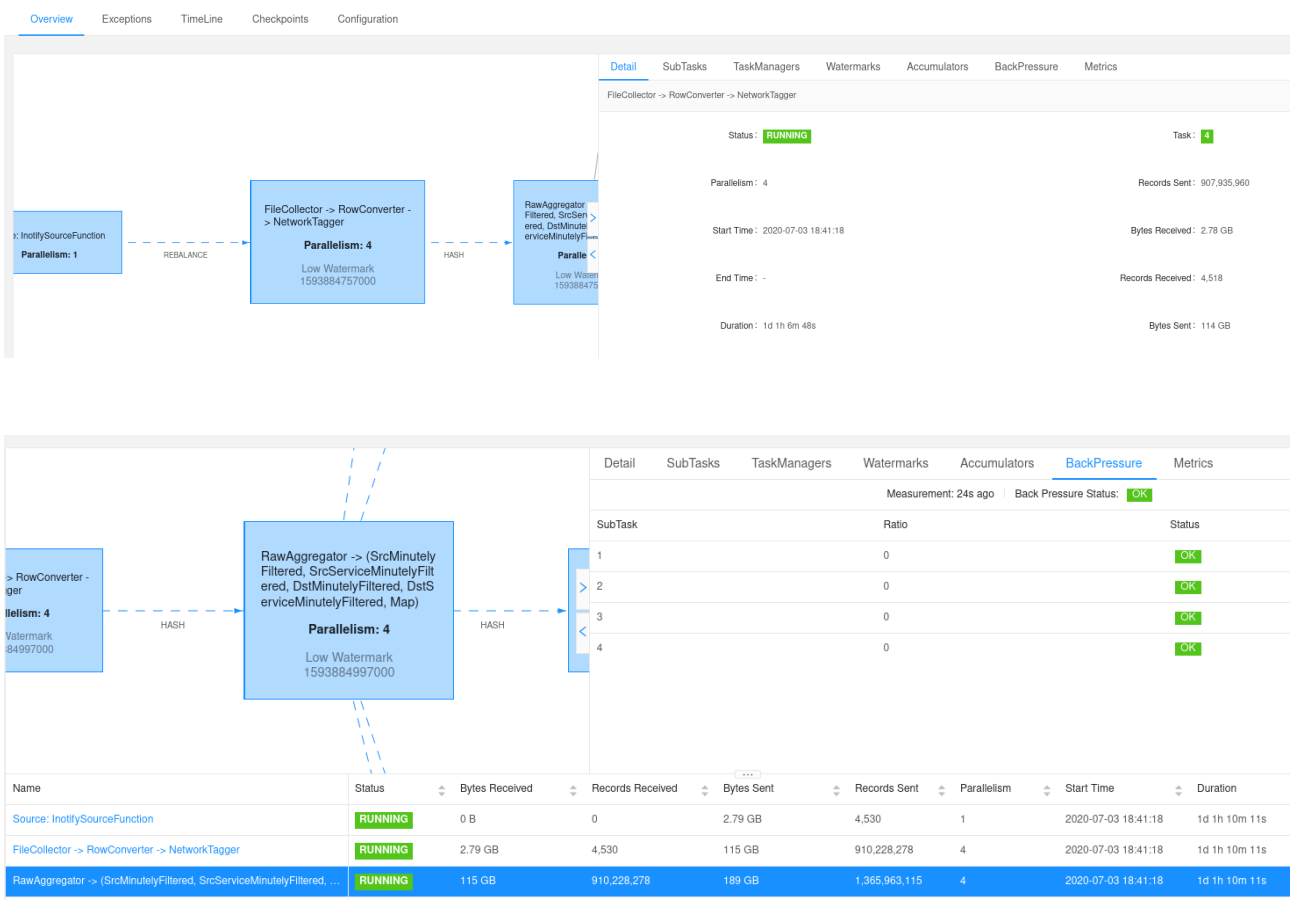

Figure B.6: Operator-level monitoring capabilities.

Glossary

Apache Flink A framework and distributed processing engine for stateful computations over data streams (https://flink.apache.org/). 10

Apache Kafka A distributed streaming platform. It is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies (https://storm.apache.org/). 10

Apache Spark A unified analytics engine for large-scale data processing. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs (https://spark.apache.org/). 10

Apache Storm A free and open source distributed realtime computation system. It makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing (https://storm.apache.org/). 10

AWK A procedural scripting programming language specialized for textual data manipulation. It was designed to accomodate one-liner programs, but it is Turing complete. 1

bubble sort Bubble sort is a sorting algorithm of average case complexity of $O(n^2)$. 26

C++ An object oriented programming language, successor of C, with one of the widest usage of the planet and an incredible range of programming capabilities. 1

ERP Enterprise resource planning: paradigm, traditionally used for batch processing. 9

ETL Extract–Transform–Load paradigm, traditionally used for batch processing. 9

FreeMarker a template engine: a Java library to generate text output of any kind based on templates and changing data. It uses a custom made template language (https://freemarker.apache.org/). 40

Go A relatively new modern programming language inheriting from C, aimed at the simplification of development of large scale programs requiring complex builds and concurrent execution. 40

Grafana An open source analytics and monitoring platform with a number of database integrations and interactive visualizations (https://grafana.com). 2, 8, 30

Inotify A Linux kernel subsystem that acts to extend filesystems to notice changes to the filesystem, and report those changes to applications. 15

Inotify tools A software package of command line utilities related to the Inotify system, including `inotifywait` and `inotifywatch` (https://github.com/inotify-tools/inotify-tools). 15

Java An object oriented programming language, with a simple syntax and famous for its ease of deployment and far reach. 10

Java Streams A sequence of elements supporting sequential and parallel aggregate operations, provided since Java 8, introducing functional capabilities to the language. 26

JavaScript A complicated interpreted programming language that embeds object oriented, procedural and functional paradigms that was designed to be run on the browser, but has since become widely used everywhere. 39

lambda A lambda expression is a function. In modern languages, it usually refers to a one-line function where the expression is the value that will be returned by the function. 23

Naudit High Performance Computing and Networking A network analysis and tools provider company based in Madrid and Pamplona. 1

NetFlow A Cisco technology that efficiently provides the metering base for a key set of applications including network traffic accounting, usage-based network billing, network planning, as well as Denial Services monitoring capabilities, network monitoring, outbound marketing, and data mining capabilities for both service provider and enterprise customers. 6, 18

P2 A Naudit HPCN technology for the generation of statistics at the IP-to-IP conversation level. 5, 15

PL/pgSQL A loadable procedural language for the PostgreSQL database system and can be used to create functions and triggers, adds control structures to the SQL language, can perform complex computations, (https://www.postgresql.org/docs/current/plpgsql.html). 8, 32

PostgreSQL A powerful, open source object-relational database system with over 30 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance (https://www.postgresql.org/). 7, 28

Prometheus a systems and service monitoring system. It collects metrics from configured targets at given intervals (https://prometheus.io/). 11, 15, 35

Python A fairly simple interpreted object oriented programming language which can be read almost as pseudocode, with extreme popularity. 1, 10

quick sort Quick sort is a sorting algorithm of average case complexity of O(n log n). 26

Record The data representation class modelling P2's output. 5, 17, 18

Ruby A fairly simple interpreted dynamic programming language with an almost natural langauge syntax. 10

Scala A modern programming language that fuses the object oriented and functional paradigms, designed to run in the JVM. 10

SQL Structured Query Language: an inmensely popular database query language. 1

systemd systemd is a suite of basic building blocks for a Linux system. It provides a system and service manager that runs as PID 1 and starts the rest of the system (https://systemd.io/). 5

time series Timestamped data that collectively represents how a system, process or behavior changes over time. 7, 31

TimescaleDB A time series SQL database providing fast analytics, scalability, with automated data management on a proven storage engine, built on PostgreSQL (https://www.timescale.com/). 7

trie A data structure also called digital / prefix tree used for search algorithms.. 19

Acronyms

CIDR Classless Inter-Domain Routing. 19

CSV Comma Separated Values. 19

JSON JavaScript Object Notation. 18

JVM Java Virtual Machine. 10

LPM Longest Prefix Match. 19

POJO Plain Old Java Object. 36

RTT Round Trip Time. 2, 18

YAML YAML Ain't Markup Language. 15

Bibliography

[Coc05] Alistair Cockburn. Hexagonal architecture. http://alistair.cockburn.us/Hexagonal+architecture, 2005. (Dead link, consult Web Archive).

[Fou19a] Apache Software Foundation. Flink documentation. https://ci.apache.org/projects/flink/flink-docs-release-1.10/, 2019.

[Fou19b] Apache Software Foundation. Using non jvm languages with storm. https://storm.apache.org/releases/2.1.0/Using-non-JVM-languages-with-Storm.html, 2019.

[KH19] Vasiliki Kalavri and Fabian Hueske. *Stream Processing with Apache Flink*. O'Reilly Media, Inc, 2019.

[Ora19] Oracle. Java documentation - the reflection api. Official website, 2019.